Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1997-03

Cross model access in the multi-lingual, multi-model database management system.

Anastasopoulos, Achilles

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/8178

Downloaded from NPS Archive: Calhoun

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NPS ARCHIVE
1997.03
ANASTASOPOULOS, A.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

CROSS MODEL ACCESS IN THE MULTI-LINGUAL, MULTI-MODEL DATABASE MANAGEMENT SYSTEM

by

Achilles Anastasopoulos

March 1997

Thesis Advisor: C. Thomas Wu
Second Reader: David K. Hsiao

Thesis
A44214

Approved for public release; distribution is unlimited.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE CROSS MODEL ACCESS IN THE MULTI-LINGUAL, MULTI-MODEL DATABASE MANAGEMENT SYSTEM	5. FUNDING NUMBERS
6. AUTHOR(S) Achilles Anastasopoulos	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Relational, hierarchical, network, functional, and object-oriented databases support its corresponding query language, SQL, DL/I, CODASYL-DML, DAPLEX, and OO-DML, respectively. However, each database type may be accessed only by its own language. The goal of M^2DBMS is to provide a heterogeneous environment in which any supported database is accessible by any supported query language. This is known as cross model access capability.

In this thesis, relational to object-oriented database cross model access is successfully implemented for a test database. Data from the object-oriented database EWIROODB is accessed and retrieved, using an SQL query from the relational database EWIROODB. One problem is that the two interfaces (object-oriented and relational) create *catalog* files with different formation, which makes the cross-model access impossible, initially. In this thesis the relational created catalog file is used, and the cross model access capability is achieved.

The object-oriented catalog file must be identical with the relational one. Therefore, work yet to be done is to write a program that automatically reformats the object-oriented catalog file into an equivalent relational catalog file.

14. SUBJECT TERMS Multi-Lingual, Multi-Model DBMS, SQL, DL/I, CODASYL-DML, DAPLEX, OO-DML, M^2DBMS, EWIROODB	15. NUMBER OF PAGES 82
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500
(Rev. 2-89)

Standard Form 298
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

CROSS MODEL ACCESS IN THE MULTI-LINGUAL, MULTI-MODEL DATABASE MANAGEMENT SYSTEM

Achilles Anastasopoulos
Lt, Hellenic Navy
B.S., Hellenic Naval Academy, 1983

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE
IN
COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1997

NPS ARCHIVE
1997.03
ANASTASOPOULOS, A.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

ABSTRACT

Relational, hierarchical, network, functional, and object-oriented databases support its corresponding query language, SQL, DL/I, CODASYL-DML, DAPLEX, and OO-DML, respectively. However, each database type may be accessed only by its own language. The goal of M^2DBMS is to provide a heterogeneous environment in which any supported database is accessible by any supported query language. This is known as cross model access capability.

In this thesis, relational to object-oriented database cross model access is successfully implemented for a test database. Data from the object-oriented database EWIROODB is accessed and retrieved, using an SQL query from the relational database EWIROODB. One problem is that the two interfaces (object-oriented and relational) create *catalog* files with different formation, which makes the cross-model access impossible, initially. In this thesis the relational created catalog file is used, and the cross model access capability is achieved.

The object-oriented catalog file must be identical with the relational one. Therefore, work yet to be done is to write a program that automatically reformats the object-oriented catalog file into an equivalent relational catalog file.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. INTEGRATION AND INTEROPERABILITY OF HDBMS 3

B. WHAT IS M^2DBMS ? 6

C. THE OBJECTIVE OF THE THESIS 6

D. THE ORGANIZATION OF THE THESIS 6

II. AN OVERVIEW OF M^2DBMS 9

A. WHY A MULTI-MODEL, MULTI-LINGUAL DBMS? 10

B. THE MULTI-LINGUAL DATABASE SYSTEM 11

C. THE MULTIBACKEND DATABASE SUPERCOMPUTER 13

III. THE RELATIONAL DATABASE INTERFACE 15

A. BASIC RELATIONAL DATABASE TERMINOLOGY 15

B. DESCRIPTION OF THE RELATIONAL DATABASE INTERFACE 21

C. LIMITATIONS 32

IV. THE OBJECT-ORIENTED DATABASE INTERFACE 35

A. BASIC OBJECT-ORIENTED DATABASE TERMINOLOGY 35

B. DESCRIPTION OF THE OBJECT-ORIENTED DATABASE INTERFACE . 40

C. LIMITATIONS 50

V. THE CROSS-MODEL ACCESS 53

A. PROBLEMS 53

B. DESCRIPTION 55

VI. CONCLUSION 59

APPENDIX 61

LIST OF REFERENCES 69

INITIAL DISTRIBUTION LIST 71

LIST OF FIGURES

1. A Heterogeneous Environment with Interoperability 3
2. A General Integrated HDBMS System 5
3. Cross Model Access Capability in M^2DBMS from Relational to Object-Oriented Database 7
4. The M^2DBMS System 12
5. The M^2DBMS Hardware Organization 14
6. An Example of a Relational Database 16
7. A Relational Database Schema 18
8. The ER Diagram 18
9. Types of Relationship Between Two Relations A and B 19
10. An Example of a Query in SQL 20
11. The Relational Database Schema of the EWIROODB Database 21
12. A Relational Database Instance of the EWIROODB Schema 22
13. The EWIROODB Database Schema Specification 24
14. The EWIROODB Template File 25
15. The EWIROODB Descriptor File 26
16. The EWIROODB Record File 28
17. The EWIRsqlreq Query File 30
18. Examples of Objects 36
19. An Example of Class "Car" 37
20. An Example of a Relationship 1:1 between the Classes "Car" and "Reservation" 38
21. An example of inheritance: Superclass "Vehicle" with Subclasses "Car" and "Bike" 39
22. The Object-oriented Database Schema of the EWIROODB Database 41
23. The Instances of the Antenna Class in the EWIROODB Object-oriented Database 41
24. The EWIROODB Database Schema Specification 43
25. The EWIROODB Template File 44

26. The EWIROODB Descriptor File 44
27. The EWIROODB Record File 46
28. The Relational .EWIROODB.cat File 54
29. The Object-oriented .EWIROODB.cat File 54

I. INTRODUCTION

One of the most common applications in the computer domain, with a huge growth during the last three decades is the area of databases. By **data** we mean known real-word facts or information that can be recorded and that have understandable meaning. A collection of data with an association among them is a **database**.

A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested [Ref. 1].

A computer **database management system (DBMS)** is a software system that allows us to create, manage, and maintain one or more databases that are stored in a computer disk.

A **distributed DBMS (DDBMS)** consists of several databases and DBMS that are connected with computer networks. Here are a few of the common uses of database management systems:

- Managing mailing lists and telephone directories.
- Managing customer, sales, and membership information files
- Managing orders and controlling inventory
- Storing and updating employee information
- Handling bookkeeping and accounting tasks

There are many different types of databases. Each type of database is defined by a specific **data model** and a corresponding **data language**. The data model provides the user with a way to specify the structure and form of the data to be stored in the database , as well as a collection of the types of general operations that are used to access the database. The data language of the database provides the user with a way to specify database operations

that are used to access the stored data [Ref. 2]. Typical manipulations include retrieval, insertion, deletion, and modification of the data.

The main factor we consider in a DBMS is the **degree of homogeneity.** If all components (servers, DBMS s, clients) use identical software, the DDBMS is called **homogeneous;** otherwise, it is called **heterogeneous.** Database systems that consist of databases with different data models are heterogeneous.

Heterogeneous database systems are very common and wide spread today, especially in large organizations, governmental environments, and computer networks. There are several reasons for this [Ref. 3, 4, 5]:

- The proliferation of different database management systems, and databases.
- The proliferation of a diversity of microcomputers and personal computers (several small databases).
- The advances in data communications and computers networks.
- The wide use of distributed databases, and the geographically distribution of the database applications of some organizations (banks and companies for example).
- The entire information requirement is too large to be maintained by a single DBMS. There is a lack of overall database planning and control.
- Several databases were developed separately for historical reasons in the same organization. Vendors of different applications supplied different DBMS packages that were incompatible with each other.
- The necessity of managing heterogeneous databases such as linking heterogeneous databases via the World Wide Web (WWW), organizing them into database federations or multidatabase systems, and constructing data warehouses. *Federated databases* consist of a collection of database systems connected in order to share and exchange information. *Data warehouse* is an application, that accumulates into one database huge amounts of information about an organization's operations, and provides the users with easy-to-use and powerful query tools with the capability of retrieving and updating individual records extremely fast.

An example of a heterogeneous environment is shown in Figure 1: a company which during the last twenty years has used a hierarchical database for product assemblies, a network database for inventory control, and two relational databases for record keeping.

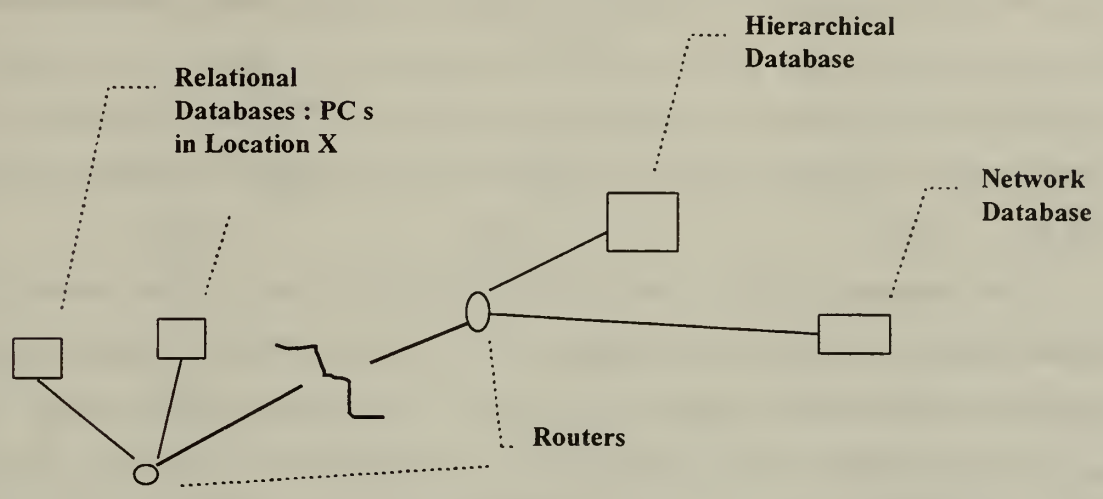


Figure 1. A Heterogeneous Environment with Interoperability

A. INTEGRATION AND INTEROPERABILITY OF HDBMS

The use of heterogeneous database systems implies the creation of problems due to the existence of different data models, data languages, and possible data incompatibilities among the existing databases. Some of the data incompatibilities are:

- Attributes of entities are stored with different units (for example: attribute "weight" is stored in kilograms in one system and in pounds in an other).
- Entities are stored in different way: two or more entities in one system are combined into one entity in an other.
- Dissimilarity in naming objects: the same name in different database systems may represent different objects, and different names may represent the same object (this is called **semantic heterogeneity**).

- Different values of attributes that refer to the same object, but are stored in different databases.

Each kind of database in an HDBMS has its own schema, expressed in its own data model, and can be accessed only by its own retrieval language. The need for integration of the several databases of a system is obvious. **Integration** is a means of combining or interfacing data and functions of a system into a cohesive set. The goal of integration is to provide access to data that is stored in different forms and managed by different systems [Ref. 6].

The integration has the purpose of "hiding" the heterogeneity of the system from the users and provides them with transparency. The user does not need to worry about the several data models, data languages, and semantics of the components of the system. The user only cares about a single interface when he communicates with the system. The user "sees" a **global schema**, and uses **a global manipulation language**. In the integrated system the different data schemata and data languages are transformed into the global schema, and the global manipulation language. The Figure 2 illustrates a general integrated system.

Due to the above mentioned problems of HDBMS the integration may be difficult to do and sometimes may not be achieved at all.

Another approach to solving the problem of proliferation of HDBMS 's is that the databases of a multidatabase system should become **interoperable**: This means that a user can access any database of the system, by using only one database (the most familiar to him / her). Applications can execute using the data language of one kind of database to access a database of a different model. This capability is also known as **cross model access**. There is no need for integration and global schema. The several databases retain their autonomy and continue to operate independently. Some of the benefits of the interoperability are data sharing, and reusability of code and transactions [Ref. 7].

An example of a heterogeneous environment with interoperability is shown in Figure 1: A company which uses one hierarchical database for product assemblies, one

network database for inventory control, and two relational databases for record keeping in a different locations. The system provides a user the ability to access any database as if were managed under any one of the four DBMS at one central location. Thus, a user could have access to any database through a relational view at one of the PC is using SQL.

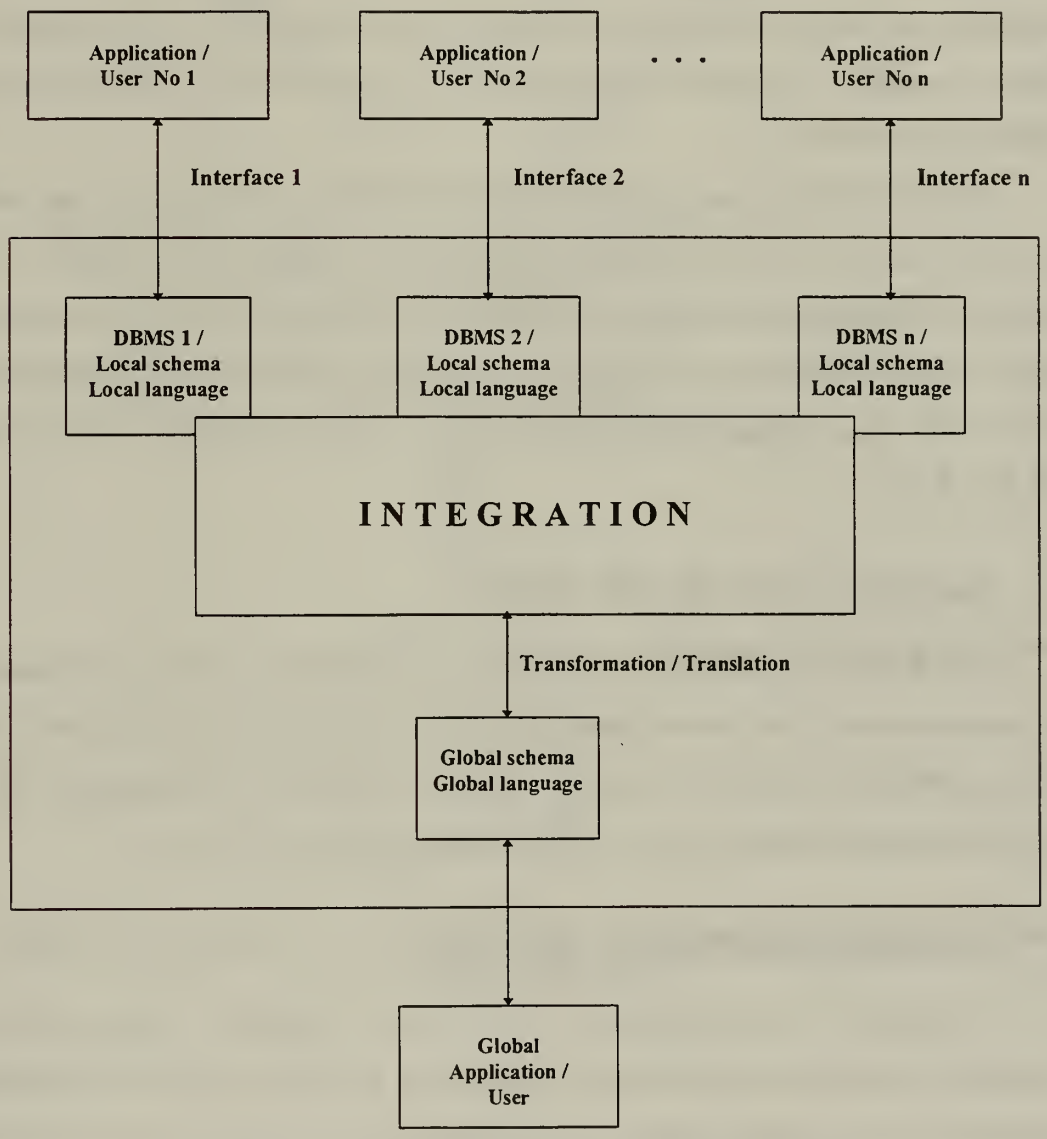


Figure 2. A General Integrated HDBMS System

B. WHAT IS M^2DBMS ?

In the NPGS Laboratory for Database Systems Research, researchers have been experimenting with a multi-database system prototype called M^2DBMS (Multi-Lingual, Multi-Model DBMS). The system supports heterogeneous databases, each of which is based on a different data model. The system executes transactions of the data language corresponding to each data model supported. So far, relational, hierarchical, network, functional, and object-oriented databases have been implemented. Correspondingly, this system is capable of executing transactions written in SQL, DL/I, CODASYL-DML, DAPLEX, and OO-DML.

The M^2DBMS supports multiple databases not as a collection of separate systems, but with a single kernel data model and language called attribute-based DBMS. All the supported heterogeneous databases are organized internally on the basis of the kernel data model. All the heterogeneous transactions are translated into their equivalent transactions in the kernel data language[Ref. 8]. Description of the M^2DBMS can be found in [Ref. 2, 8, 9, 10].

C. THE OBJECTIVE OF THE THESIS

The M^2DBS system supports databases with different data models and languages. We want to develop a cross-model access to get the desired interoperability, where the users can access the same database with either relational, or object-oriented interface. Our first goal is to provide relational access to object - oriented database (Figure 3).

D. THE ORGANIZATION OF THE THESIS

In Chapter II of this thesis, an overview of the M^2BMS is given, with a brief description of the hardware and software organization. In Chapters III and IV the relational and object-oriented interfaces and their limitations are described. In chapter V the effort for the cross-model access and problems are covered. In Chapter VI, conclusions are given.

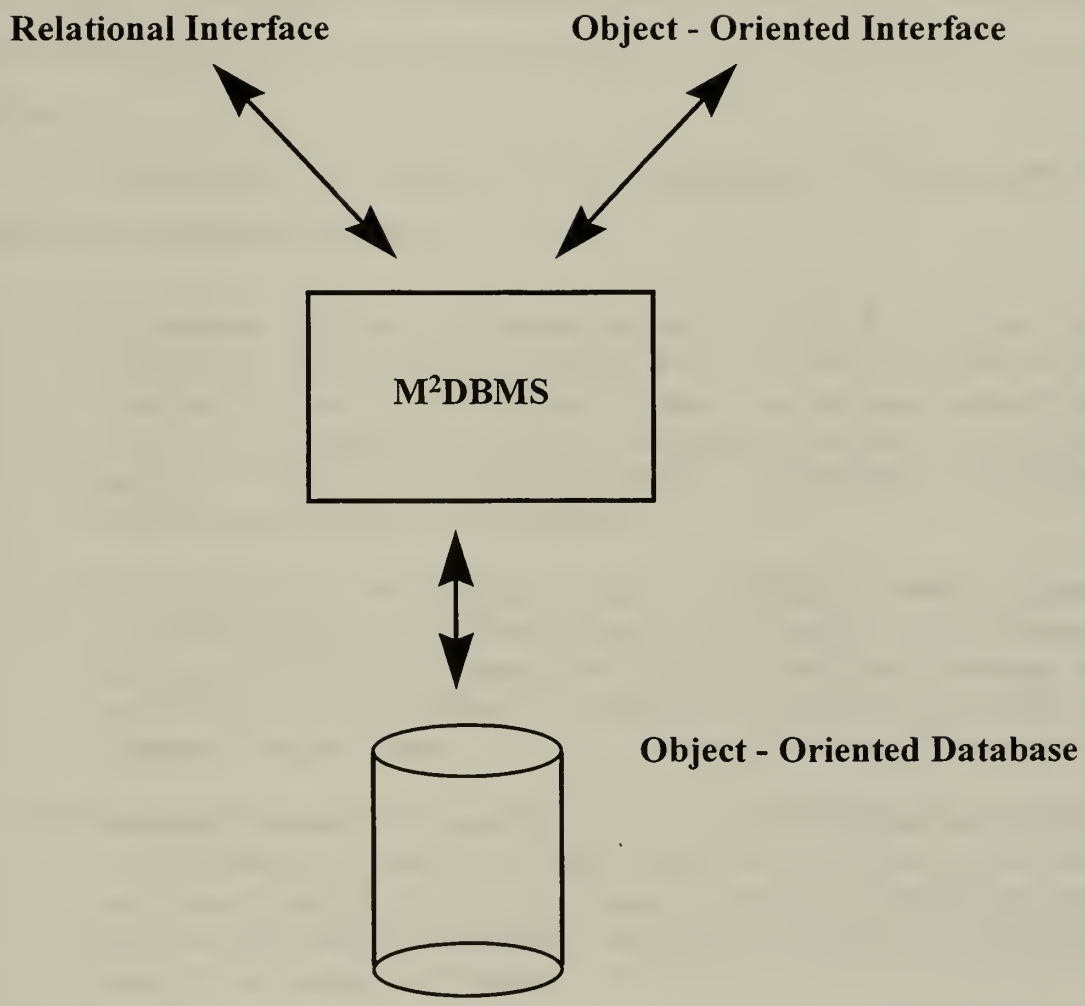


Figure 3. Cross Model Access Capability in M^2DBMS from Relational to Object-Oriented Database

II. AN OVERVIEW OF M^2DBMS

Unlike traditional DBMS which are **monomodel** and **monolingual** (support only one kind of database), the M^2DBMS is **multimodel** and **multilingual** [Ref. 8]. This means that it supports the meaning of the schemata of databases created under different data models as well as the execution of transactions written in different data languages on these databases.

The different kinds of databases and their corresponding data languages that are supported by the system are as follows:

- **Hierarchical** database with transactions written in **DL/I**. The basic data structures of the hierarchical database are the *records* and *parent-child relationships.* A record is a collection of values that give information for an entity or a relationship. Parent-child relationships describe a relationship between two records. The hierarchical database was developed in the sixties mainly for supporting product assemblies

- **Network** database with transactions written in **CODASYL-DML**. The basic data structures of the network database are the *records* and *sets.* Records consist of a group of related data values. Sets describe a relationship between two records. The network database was developed in the seventies mainly for supporting inventory control.

- **Relational** database with transactions written in **SQL**. The basic data structure of the relational database is the *table.* Each row in the table represents a collection of related data values. These values are data that describe a real-world entity or relationship. The relational database was developed in the eighties mainly for supporting record keeping.

- **Functional** database with transactions written in **DAPLEX**. The basic data structure of the functional database are the *entities* (corresponding to real-world objects) and *functional relationships.* The functional database was developed in the eighties primarily for supporting a lot of facts and rules for making inferences.

- **Object-oriented** database with transactions written in **OO-DML**. The basic data structure of the object-oriented database is the *object.* An object model represents a real-world entity with its behavior and interactions. The object-

oriented database was developed in the nineties mainly for supporting the object technology.

A. WHY A MULTI-MODEL, MULTI-LINGUAL DBMS?

The existence of heterogeneous database systems is very common today. Two crucial issues for the effective and efficient utilization of the HDBMS are **data sharing** and **resource consolidation** [Ref. 8, 9]. The M^2MDBS addresses both issues of data sharing and resource consolidation while maintaining the autonomy of the individual databases [Ref. 8].

Data sharing has a direct association with the interoperability, which is the ability that allows the users of the system to access the different databases of the system with transactions written only in one data language of the system. Since there is not a need to translate a transaction written in one data language to another data language, data sharing implies reusability of code and transactions. The system provides reduction in data duplication and storage requirements since we keep only one copy of the data in one data model without keeping the same data in another database. Since we do not need to keep the same record in different databases, a possible change of this record kept in one database does not require updating of the same record in other databases.

Resource consolidation has a direct relationship with the integration of heterogeneous databases in a HDBMS. Resource consolidation is the combining of multiple entities executing the same functions in a database management system. Resources to be consolidated are heterogeneous databases, software, hardware, and the support personnel. Resource consolidation provides the organization a reduction of the whole cost. Since there is only a single multilingual, multimodel database computer to maintain, instead of five or six separate computers for example, the costs associated with buying software and hardware (upgrading with new versions for example), and providing support (manuals, administrators, technicians) are extensively reduced. Also, we do not need to train the users to operate different databases with different interfaces, since the system provides the users

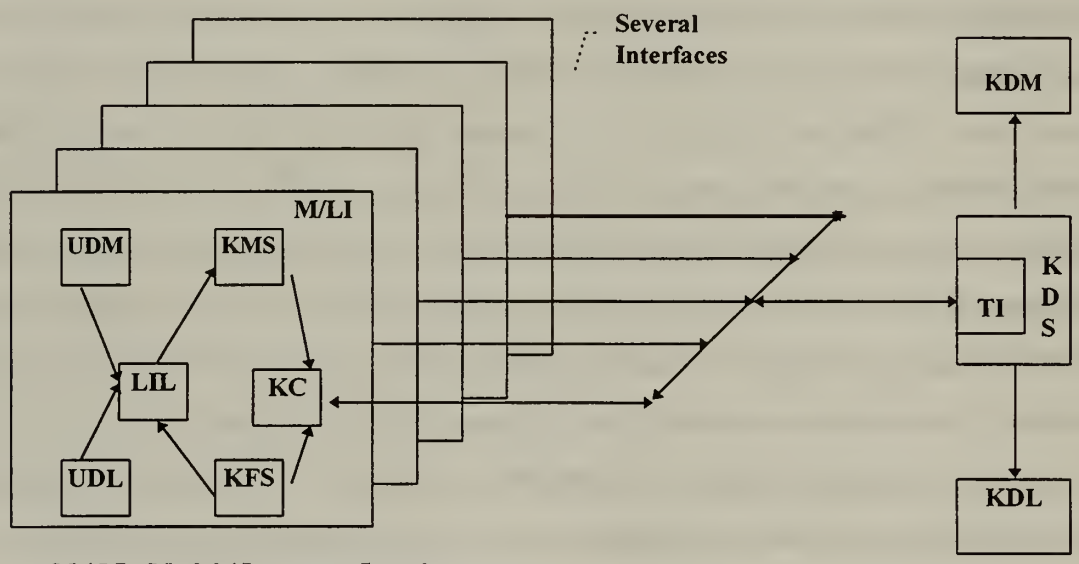
with a analogous interface for each of the different databases. There are no new system features to learn.

B. THE MULTI-LINGUAL DATABASE SYSTEM

In the multi-model, multi-lingual database computer there are currently five data models and data languages that must be transformed into a single data model and language. The transformed databases and transactions can then be processed and executed by the multi-lingual computer. These transformations are also referred to as *mappings* in the M^2DBMS. There are two different types of mappings in the M^2DBMS, the data-model transformation and the data-language translation. *Data-model transformation* is the process which takes a database modeled in one form and transforms it to an equal database in another form (kernel database). *Data-language translation* is the process which takes an operation in one data language and translates it into an equivalent operation in another data language (kernel language). The kernel data model and its kernel data language used by M^2DBMS is the *attribute-based data model (ABDM)* and the *attribute-based language (ABDL).* The attribute-based data model and language offer a complete set of means for defining and accessing databases. The ABDM supports he five basic database operations: *Retrieve, retrieve common, insert, update,* and *delete.* The user is not aware of the kernel data model and kernel data language transformations. The user needs to know only the interface of the specific type of database he/she uses.

In Figure 4, the modules, which are used for the mapping of data models and languages into kernel data models and languages, are shown. The four main modules, that are known as the **model language interface**, are the **language interface layer (LIL)**, the **kernel mapping system (KMS), the kernel controller (KC),** and the **kernel formatting system (KFS),** and they correspond to each data model . The user interacts with the system using the LIL which corresponds to the chosen **user data model (UDM)** to issue transactions that are written in the corresponding user **data language (UDL)**. LIL routes the user transactions to the KMS. The user interacts with the system using the LIL which

corresponds to the chosen **user data model (UDM)** to issue transactions that are written in the corresponding user **data language (UDL)**.LIL routes the user transactions to the KMS. The other task of the KMS is **data language translation**. KMS translates the UDL transactions into the corresponding **kernel data language (KDL)** transactions. KMS routes the KDL transaction to KC which in turn send it to KDS for execution. After completion of execution, KDS sends the results in KDM form back to the KC. KC routes the results to the KFS which in turn formats the results from KDM form to UDM form. KFS send the results to the user through LIL.



M / LI : Model / Language Interface

LIL : Language Interface Layer

KMS : Kernel Mapping System

KC : Kernel Controller

KFS : Kernel Formatting System

TI : Test Interface

KDM : Kernel Data Model

KDS : Kernel Database System

KDL : Kernel Data Language

UDM : User Data Model

UDL : User Data Language

Figure 4. The M^2DBMS System

C. THE MULTIBACKEND DATABASE SUPERCOMPUTER

The M^2DBMS uses multiple *backends* processors connected in parallel to a single controller (see Figure 5). Each backend has its own hardware , software, and disk system. All the backends computers are controlled by a backend controller (micro-processor based computer). The controller is responsible for interfacing between the backends computers and the users and hosts. The controller receives user requests in the form of database transactions, and transmits them simultaneously to the backends computers. The backends computers perform the requested database operations on the database which is distributed across the disk systems. Results from the database operation are forwarded to the controller, which in turn send them to the host. The controller and processors are connected by a communication bus (Ethernet cable).

The multiple backend architecture provides performance gain to the system. For a given size of database and a given query there is a reduction in the response time when the number of backends is increased. The system also is expandable along with the addition of more parallel backends computers without the development of any new hardware or modification of the existing hardware or software.

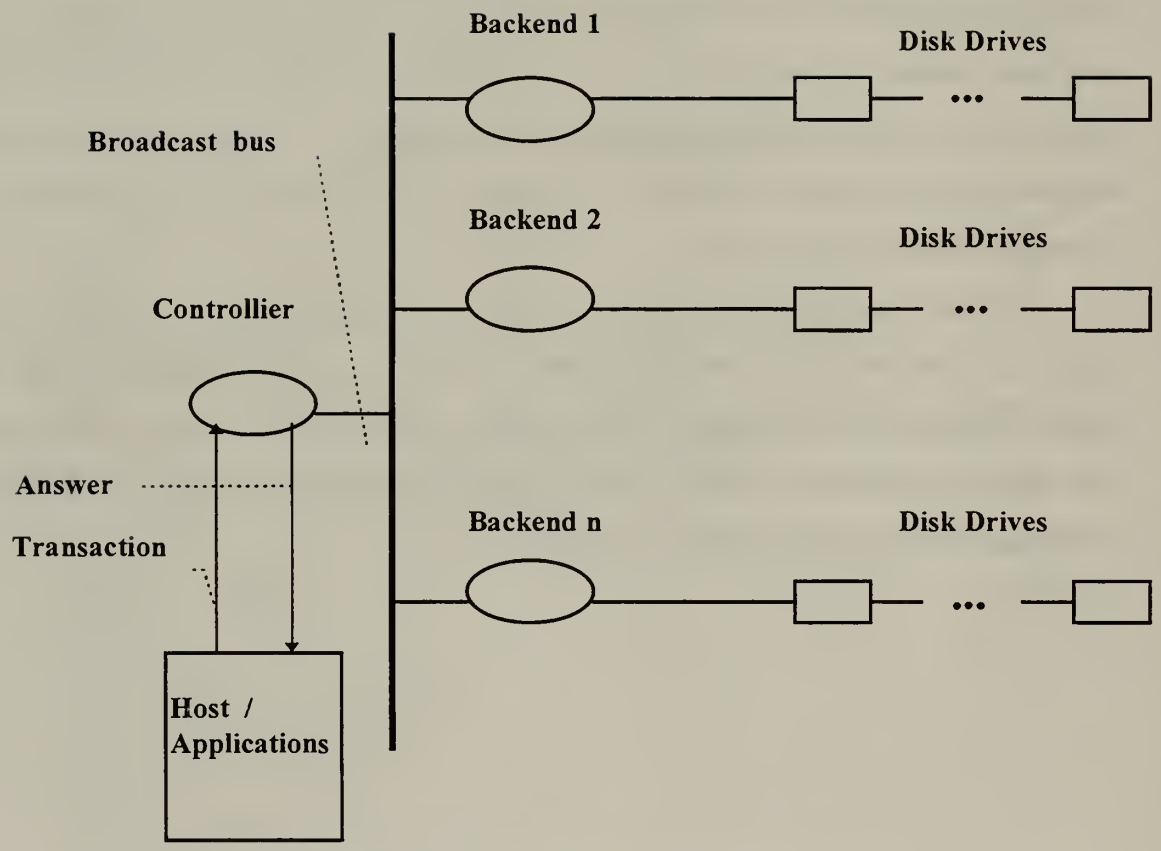


Figure 5. The M^2DBMS Hardware Organization

III. THE RELATIONAL DATABASE INTERFACE

Relational database systems are based on *the relational model of data*, first proposed by Codd in 1970 [Ref. 14]. The relational model is the way of looking and representing data that represent real world objects and their relationships.

A. BASIC RELATIONAL DATABASE TERMINOLOGY

In a relational management database system, the database is perceived by the user as a collection of *tables* (or *relations*). A relation represents relevant data relating to one type of real world object or entity (product, customer, order, for example). Figure 6 illustrates a very simple relational database referring to a sales company. The relation PRODUCT contains information (product code, description, price, etc.) about the current inventory of products . The relation CUSTOMERS keeps information (customer no., last name, first name, address, etc.) about the customers. The relation ORDER contains information (customer, product, quantity, date, etc.) about orders.

Each column of the table stands for an attribute of the relation. An *attribute* represents one type of data relating to a relation (product code, product description, customer last name, for example). Each row of the table stands for a tuple of the relation. A *tuple* is a collection of data describing one real world object or entity, an instance of a relation (a specific product <564-987, Calculator, 19.99 > for example).

A *domain* is a conceptual set of values, from which one or more columns, in one or more tables, draw their actual values. A given domain contains all permitted and possible values of some particular type. The values in a domain are generally assumed to be *atomic*, which means that they have no internal structure (they are indivisible), so far as the DBMS is concerned. An example for a domain is the range 0 - 65 for the possible values of age of the employees of a company.

PRODUCT	Product_Code	Description	Unit price $
	508-234	Pocket Organizer	69.99
	567-089	Spelling Corrector	45.99
	564-987	Calculator	19.99
	555-983	Printer	349.99
	467-922	Fax	224.99

Relations

Attributes

CUSTOMER	Cust_No	Last Name	First Name
	201	Gomez	Mary
	202	Williams	Ann
	203	Kakis	Lou
	204	Smith	Gus
	205	Moutos	Eddy

Tuples

ORDER	Invoice #	Cust_No	Product_#	Date	Quantity
	1001	201	508-234	4-6-96	2
	1002	201	567-089	3-7-96	4
	1003	203	564-987	3-1-96	1
	1004	204	555-983	1-12-97	1
	1005	202	467-922	1-15-97	5

Figure 6. An Example of a Relational Database

Here is a definition (taken from [Ref. 11]), of the term "relation:"

A relation R on a collection of domains D1, D2 , ... , Dn (not necessarily all distinct) consists of two parts, a *heading* and a *body*.

- The heading consists of a fixed set of distinct attributes {A1, A2, ... , An}, or more precisely attribute-domain pairs, { (A1:D1), (A2:D2), ... , (An:Dn) } such that each attribute Aj corresponds to exactly one of the underlying domains Dj (j = 1, 2, ..., n).

- The body consists of a time-varying set of tuples, where each tuple in turn consists of a set of attribute-value pairs { (A1:vi1), (A2:vi2), ..., (An:vin) } (i = 1, 2, 3, ... , m, where m is the number of tuples in the set). In each such tuple, there is one such attribute-value pair (Aj:vij) for each attribute Aj in the heading. For any given attribute-value pair (Aj:vij), vij is a value from the unique domain Dj that is associated with the attribute Aj.

A *relation schema* R denoted by R(A1, A2, ..., An), is made up of a relation name and a list of attributes A1, A2, ..., An , and describes the structure of a relation. An example of a relational schema for the relation PRODUCT is:

PRODUCT (Product_Code, Description, Unit price $)

The *primary key* of a relation is any set of attributes, which uniquely identifies any one tuple. The primary keys in a relational schema are underlined. An example for a key is the attribute <product_code> for the PRODUCT relation: Each product has its own unique <product_code>.

The *entity integrity constraint* states that no attributes participating in the primary key are allowed to accept null values. The *referential integrity constraint* states that a tuple in one relation that refers to another relation must refer to an existing tuple in the other relation.

Figure 7 illustrates the relational database schema of the sales. company database of the example in Figure 6, with the referential integrity constraints:

- The values of the attribute < Cust_No > in every ORDER tuple must match the < Cust_No > value of some tuple in the CUSTOMER table.
- The values of the attribute < Product_# > in every ORDER tuple must match the < Product_Code > value of some tuple in the PRODUCT table.

The *ER diagram* is a graphical notation that displays the entities, attributes, and relationships of a relational schema. Figure 8 illustrates the ER diagram of the sales company database of the example in Figure 6.

Relationships are classified into types according to the number of instances of the tuples of the related entities that can participate in. Common types for binary relationships

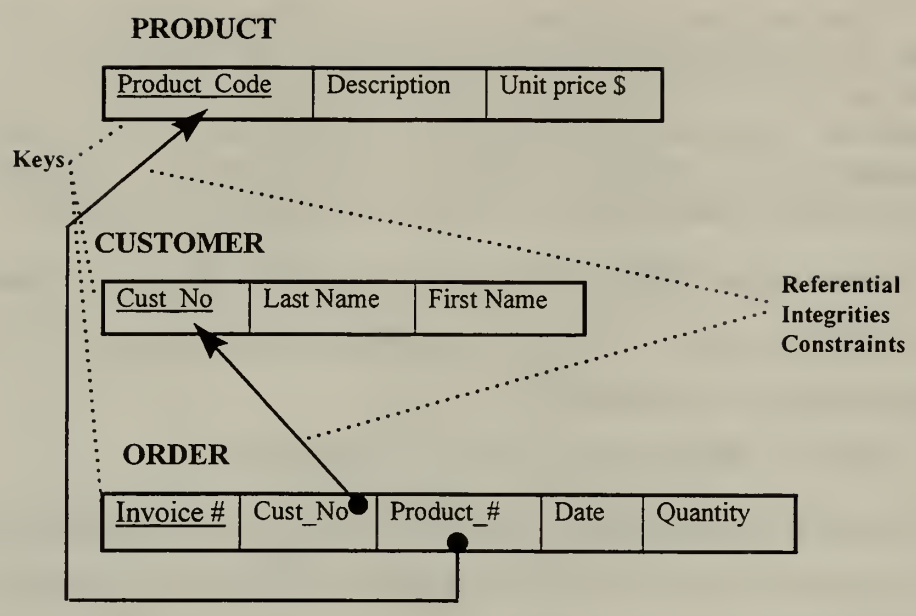


Figure 7. A Relational Database Schema

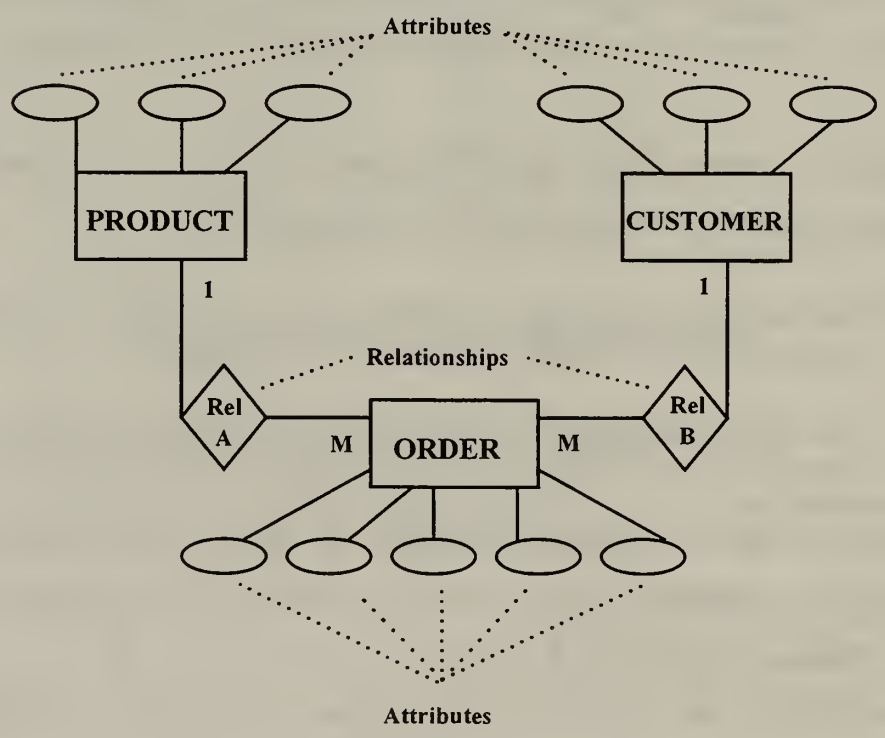


Figure 8. The ER Diagram

are 1:1, 1:M, and M:N. In the sales company example (Figure 8), the relationship A, is type 1:M, and that means that a product can relate with many orders, but an order is related with only one product. Figure 9 illustrates the possible types of relationships between two relations A and B, in reference with the number of instances of the tuples of the related entities that can participate in.

A \ B	0 or 1 (most 1)	1	M
0 or 1 (most 1)	(0 or 1) to (0 or 1)	(0 or 1) to 1	(0 or 1) to M
1	1 to (0 or 1)	1 to 1	1 to M
M	M to (0 or 1)	M to 1	M to M

Figure 9. Types of Relationship Between Two Relations A and B

When we say that a relation is "time-varying" in the relational model, we mean that, as time progress, we can insert new tuples, delete tuples, and modify the values of some attributes of the relation.

A *query* is a statement that extracts information from a database. A model is useful when there is a appropriate language for declaring queries about properties represented by the model. From the conceptual point of view, these languages are based on a simple, formal language called *relational algebra*. Relational algebra consists of a collection of operations over the relations. First, the relational algebra contains the usual set operations:

Union, intersection, difference, and Cartesian product. Second, this algebra also contains specifically developed operations for relational databases, with the most common of them:

- Select: Is used to select a subset of the tuples in a relation that fulfill a selection condition.
- Project: Is used to select certain columns from a relation.
- Join: Is used to mix related tuples from two or more relations into tuples. Join operation allows us to process relationships among relations.

The result of the queries is also a table.

A number of relational query languages have been designed and implemented to serve as practical tools for the users. The most common and powerful language is the SQL. A description of the SQL is given in [Ref. 1]. Many commercial applications for relational databases are based on the SQL. An example of use of SQL and the result for the sales company is given in Figure 10.

Retrieve the description and quantity of products that were sold in 3-1-96.

```
SELECT  DESCRIPTION, QUANTITY
FROM    ORDER, PRODUCT
WHERE   DATE = "3-1-96" AND PRODUCT_CODE = PRODUCT_#
```

Result :

Description	Quantity
Calculator	1

Figure 10. An Example of a Query in SQL

B. DESCRIPTION OF THE RELATIONAL DATABASE INTERFACE

The EWIR database was used in References [12] and [13] for reactivation of the relational interface in the M^2DBMS and for implementation of the object-oriented interface. The EWIR database provides an accurate source of information for data on radars, jammers, navigational aids, and numerous non-communication electronic emitters. In the current thesis, and for purpose of clarity for the cross-model effort, a small subset of the EWIR database is used for both relational and object-oriented interface. This subset is called EWIROODB database. The Figure 11 illustrates the relational database schema of the EWIROODB database and describes the structure of the relations. The Figure 12 illustrates the relational database instance of the EWIROODB schema that is used in the current thesis. Reference [10] provides a complete description of the interfaces of the different databases of the M^2DBMS.

A user, in order to log into the M^2DBMS, must use the **mdbs** account in the Naval Postgraduate School's Laboratory for Database Systems Research on the Multi-Backend

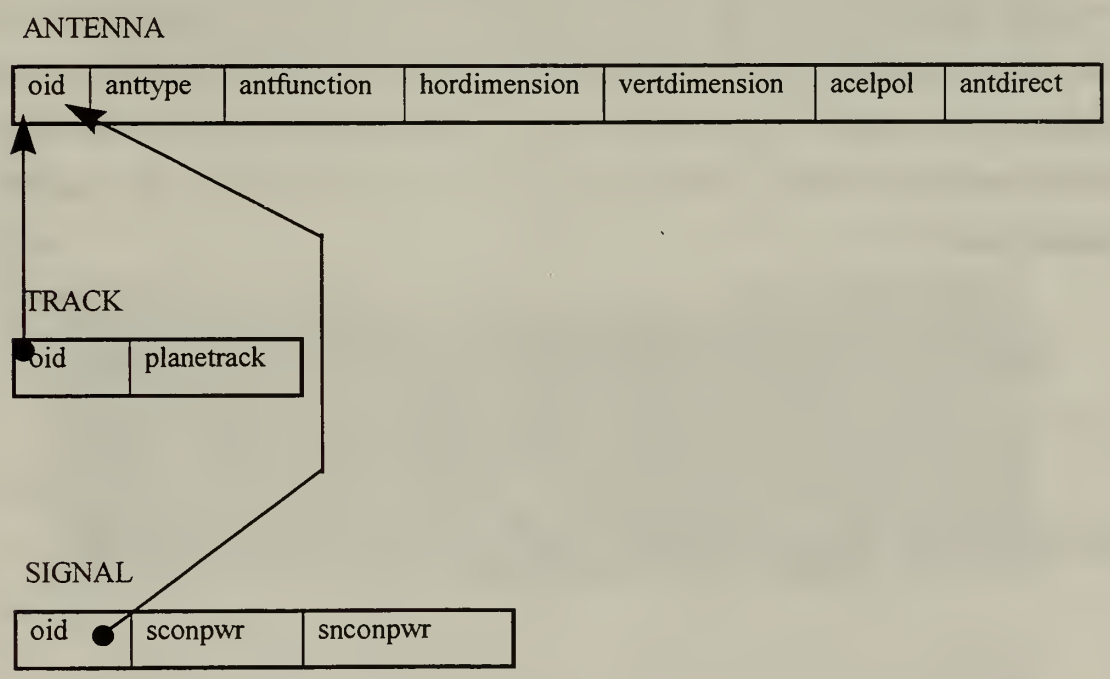


Figure 11. The Relational Database Schema of the EWIROODB Database

ANTENNA	oid	anttype	antfunction	hordimension	vertdimension	acelpol	antdirect
	Aa1	Phasedarray	Longrngaa	3ft	4ft	Rad1	Pp1
	Aa2	Squaresail	Longrngaa	3ft	4ft	Rad2	Pp2
	Aa3	Parabolic	Longrngaa	325ms	300kw	Rad2	Pp3

TRACK	oid	planetrack
	Sca1	325ms
	Sca2	300kw
	Sca3	300ms

SIGNAL	oid	sconpwr	snconpwr
	Sca1	Unidirec	128ms
	Sca2	Parabolic	Level2
	Sca3	Parabolic	Level2

Figure 12. A Relational Database Instance of the EWIROODB Schema

Database Supercomputer. Logging into terminal **db11** with the mdbs account will take the user into the default directory of **db11/u/mdbs**. At this point, the user should enter the run command (for example, **tbg** for selection of one of the existing versions of the relational interfaces). After the appropriate initiation the system prompt asks the user to select the desired interface:

```
Select an operation:
    (a) - Execute the attribute-based/ABDL interface
    (r) - Execute the relational/SQL interface
    (h) - Execute the hierarchical/DL/I interface
    (n) - Execute the network/CODASYL interface
    (f) - Execute the functional/DAPLEX interface
    (o) - Execute the Object-Oriented interface
    (x) - Exit to the operating system
```

The user, in order to proceed into the relational interface, should enter (r):

```
Select-> r
```

At this point, the system will prompt the user for the operation desired: Select option **(l)** to load e new database, or **(p)** to process a database that already is resident in the system:

```
Enter type of operation desired
    (l) - load new database
    (p) - process existing database
    (x) - return to the MLDS/MBDS system menu
```

The user in order to load a new database should enter (l):

```
Action --- > l
```

At this point, the system will prompt the user for the name of the database to be loaded and the name EWIROODB is entered:

```
Enter name of database ---> EWIROODB
```

After the user has entered the database name, the system prompt will ask the user to select the mode of input that is desired for loading the schema:

```
Enter mode of input desired
        (f) - read in a group of creates from a file
        (t) - read in creates from the terminal
        (x) - return to the main menu
```

The option (t) requires loading the schema from the terminal. The option (f) (reading from a file) is highly recommended because it is more convenient, since the schema file has already created and exists in the UserFiles directory (see Figure 13 EWIROODBsqldb File):

```
Action --- > f
```

After the user has entered the mode (f), the system prompt will ask the user to enter the name of the schema file and the name EWIROODBsqldb is entered:

```
What is the name of the CREATE/QUERY file ---> EWIROODBsqldb
```

For clarity, all schema files should be named in the following convention:

<database name><the acronym of the interface language (sql here)>db.

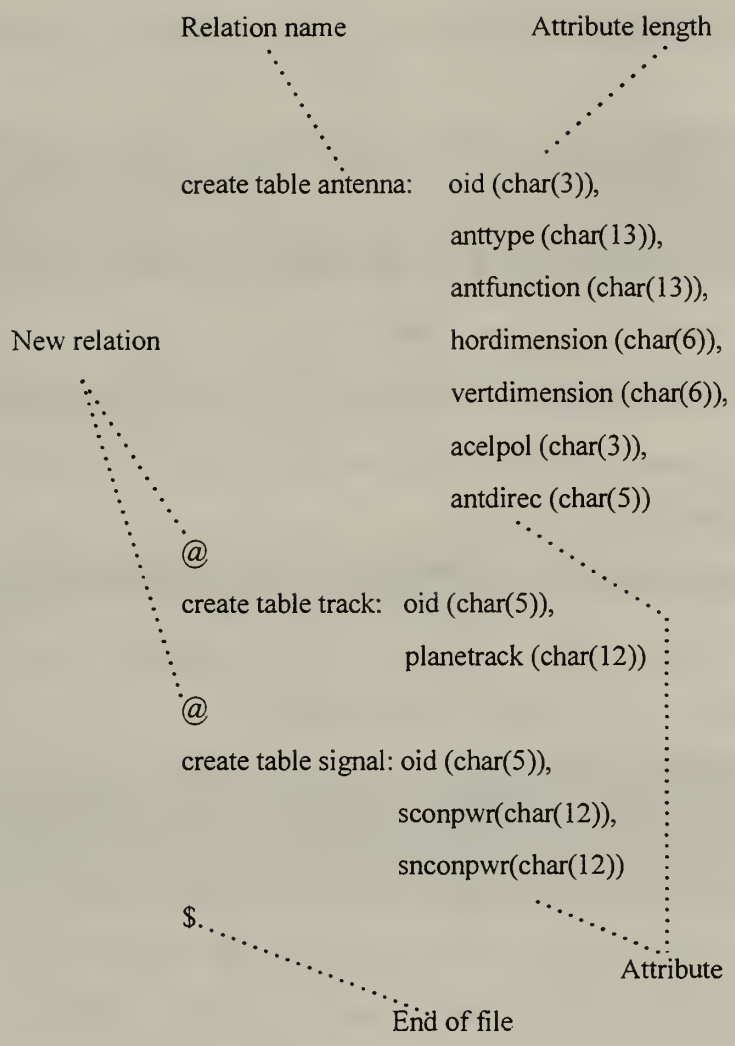


Figure 13. The EWIROODB Database Schema Specification

It is from the loading of this file that the template file (see Figure 14, EWIROODB.t file) and descriptor file (see Figure 15, EWIROODB.d file) are generated by the Language Interface Layer. The template file provides the specification of the relational database in the kernel database by creating the attribute-value pair used by the kernel system. The descriptor file provides the kernel system with a list of all the relations in the database.

EWIROODB ..... Name of the database

3 ..... Number of relations in the database

8 ..... Number of attributes in the next relation

Antenna ..... Name of a relation

TEMP s

OID s

ANTTYPE s

ANTFUNCTION s

HORDIMENSION s

VERTDIMENSION s

ACELPOL s

ANTDIREC s

3

Track

TEMP s

OID s ..... Attributes and types

PLANETRACK s

4

Signal

TEMP s

OID s

SCONPWR s

SNCONPWR s

Figure 14. The EWIROODB Template File

EWIROODB ..... Name of the database

TEMP b s

! Antenna

! Track ..... Relations in the database

! Signal

@

$

Figure 15. The EWIROODB Descriptor File

The M^2DBMS system will parse the schema file and transform the relational schema into the kernel data model language, ABDL. The parse will determine what the relational schema are and the relations are been displayed on the screen:

```
The following are the Relations in the EWIROODB Database:

        ANTENNA
        TRACK
        SIGNAL

Beginning with the first Relation, we will present each
Attribute of the relation.  You will be prompted as to whether
you wish to include that Attribute as an Indexing Attribute,
and, if so, whether it is to be indexed based on strict
EQUALITY, or based on a RANGE OF VALUES. If you do not want
to enter any indexes for your database, type an 'n' when
the Action --> prompt appears
Strike RETURN or 'n' when ready to continue.
```

The system gives the opportunity for indexing the attributes in the relations, but this option is not usually used. The user enters (n):

```
Action --- > n
```

At this point the system will prompt the user for the operation desired: Select option **(l)** to load e new database, or **(p)** to process a database that already is resident in the system:

```
Enter type of operation desired
```

```
(l) - load new database
(p) - process existing database
(x) - return to the MLDS/MBDS system menu
```

Since the database is resident now in the system and the schema exists on the M^2DBMS, the user should select now the option (p):

```
Action --- > p
```

At this point the system will prompt the user for the name of the existing database and the name EWIROODB is entered:

```
Enter name of database ---> EWIROODB
```

After the user has entered the database name, the system prompt will ask the user to select the mode of input that is desired for loading the records:

```
Enter mode of input desired
    (f) - read in a group of queries from a file
    (t) - read in queries from the terminal
    (m) - mass load a file
    (d) - display the current database schema
    (x) - return to the previous menu
```

The options (t) and (f) require the input of the records in SQL transactions, from the terminal or a file. The option (m) (mass loading from a file) is highly recommended because it is more convenient, since the record file has already created and exists in the UserFiles directory (see Figure 16, EWIROODB.r File):

```
Action --- > m
```

After the user has entered the mode (m), the system prompt will ask the user to enter the name of the record file and the name EWIROODB.r is entered:

```
Enter name of record file ---> EWIROODB.r
```

For clarity, all record files should be named in the following convention:

<database name><.r>

EWIROODB ..... Name of the database

@

ANTENNA

Aa1 Phasedarray Longrngaa 3ft 4ft Rad1 Pp1

Aa2 Squaresail Longrngaa 3ft 4ft Rad2 Pp2 ..... Tuples

Aa3 Parabolic Longrngaa 325ms 300kw Rad2 Pp3

@

TRACK

Sca1 325ms

Sca2 300kw

Sca3 300ms

@ ..... New relation

SIGNAL

Sca1 Unidirec 128ms

Sca2 Parabolic Level2

Sca2 Parabolic Level2

$ ..... End of file

Figure 16..The EWIROODB Record File

(EWIROODB.r File)

After entering the mass load file name, a sequence of ABDL insert statements will appear:

```
[INSERT (<TEMP, Antenna>, <OID, Aa1>, <ANTTYPE, Phasedarray>,
<ANTFUNCTION, Longrngaa>, <HORDIMENSION, 3ft>,
<VERTDIMENSION, 4ft>, <ACELPOL, Rad1>, <ANTDIREC, Pp1>)]
[INSERT (<TEMP, Antenna>, <OID, Aa2>, <ANTTYPE, Squaresail>,
<ANTFUNCTION, Longrngaa>, <HORDIMENSION, 3ft>,
<VERTDIMENSION, 4ft>, <ACELPOL, Rad2>, <ANTDIREC, Pp2>)]
```

```
[INSERT (<TEMP, Antenna>, <OID, Aa3>, <ANTTYPE, Parabolic>,
<ANTFUNCTION, Longrngaa>, <HORDIMENSION, 325ms>,
<VERTDIMENSION, 300kw>, <ACELPOL, Rad2>, <ANTDIREC, Pp3>)]
[INSERT (<TEMP, Track>, <OID, Sca1>, <PLANETRACK, 325ms>)]
[INSERT (<TEMP, Track>, <OID, Sca2>, <PLANETRACK, 300kw>)]
[INSERT (<TEMP, Track>, <OID, Sca3>, <PLANETRACK, 300ms>)]
[INSERT (<TEMP, Signal>, <OID, Sca1>, <SCONPWR, Unidirec>,
<SNCONPWR, 128ms>)]
[INSERT (<TEMP, Signal>, <OID, Sca2>, <SCONPWR, Parabolic>,
<SNCONPWR, Level2>)]
[INSERT (<TEMP, Signal>, <OID, Sca2>, <SCONPWR, Parabolic>,
<SNCONPWR, Level2>)]
Exit mass_load
```

At this point, the user is ready to process SQL transactions against the database that is currently residing on the system. The system prompt will ask the user to select the mode of input that is desired for reading the queries:

```
Enter mode of input desired
    (f) - read in a group of queries from a file
    (t) - read in queries from the terminal
    (m) - mass load a file
    (d) - display the current database schema
    (x) - return to the previous menu
```

The options (t) and (f) require the input of the records in SQL transactions, from the terminal or a file, correspondingly. The option (f) (reading from a file) is highly recommended because it is more convenient, since the query file is already created and exists in the UserFiles directory (see Figure 17, EWIRsqlreq File):

```
Action --- > f
```

After the user has entered the mode (f), the system prompt will ask the user to enter the name of the query file and the name EWIRsqlreq is entered:

```
What is the name of the CREATE/QUERY file ---> EWIRsqlreq
```

For clarity, all query files should be named in the following convention:

<database name><sqlreq>

```
select *
from signal
@
select *
from track                                  SQL
@                                           transactions
select *
from antenna
@
select anttype
from antenna
where anttype = 'Phasedarray'
$ ..............................................  End of file
```

Figure 17. The EWIRsqlreq Query File

After entering the query file name, the system will scan the request file and, since there are multiple transactions in the EWIRsqlreq file, will number each transaction. The system prompt will ask the user to select the number of transaction that is desired to proceed (option (num)), or to redisplay the file of queries (option (d)), or to return to the previous menu (option (x)):

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding queries
     (d)   - redisplay the file of queries
     (x)   - return to the previous menu
```

At this point the user enters the number 1, in order to proceed the transaction:

```
select *
from signal
```

which has the meaning "retrieve all the elements of the relation **signal**:"

```
Action --- > 1
```

After the number 1 has been entered, the results of the query are displayed on the screen:

```
OID  |SCONPWR     |SNCONPWR    |
      Sca1 |Unidirec    |128ms       |
      Sca2 |Parabolic   |Level2      |
      Sca2 |Parabolic   |Level2      |
```

The system prompt will ask again for the user to select the number of transaction that is desired to proceed (option (num)), or to redisplay the file of queries (option (d)), or to return to the previous menu (option (x)):

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding queries
     (d)  - redisplay the file of queries
     (x)  - return to the previous menu
```

At this point the user enters the number 2, in order to proceed the transaction:

```
select *
from track
```

which has the meaning " retrieve all the elements of the relation **track** :"

```
Action --- > 2
```

After the number 2 has been entered, the results of the query are displayed on the screen:

```
OID  |PLANETRACK  |
Sca1 |325ms       |
Sca2 |300kw       |
Sca3 |300ms       |
```

The system prompt will ask again for the user to select the number of transaction that is desired to proceed (option (num)), or to redisplay the file of queries (option (d)), or to return to the previous menu (option (x)):

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding queries
     (d)  - redisplay the file of queries
```

```
(x)  - return to the previous menu
```

At this point the user enters the number 4, in order to proceed the transaction:

```
select anttype
from antenna
where anttype = 'Phasedarray'
```

which has the meaning "retrieve all the tuples of the attribute **anttype** of the relation **antenna**, which have value '**Phasedarray**':"

```
Action --- > 4
```

After the number 4 has been entered, the results of the query are displayed on the screen:

```
ANTTYPE     |
------------
Phasedarray |
```

Having retrieved the desired data , the user exits of the system choosing the options (x).

C. LIMITATIONS

As already mentioned in the previous section, the system provides the user the option to load the schema, records, and queries via corresponding files that must have been already created and reside in the directory db11/u/mdbs/UserFiles. A subdirectory of UserFiles can be used, but in that case it must be included in the input when the user enters the name of the file (for example: What is the name of the CREATE/QUERY file ----> /relational/EWIROODBsqldb)

For clarity, all files should be named in the following convention:

<database name>sqldb for the schema files,

<database name>.r for the record files,

and *<database name><sqlreq>* for the query files.

The use of <*database name*> is recommended in order the files of a database can be easily distinguishable among several developed databases.

All files must have a dollar sign "$" on the last line of the file to signal the end of the file, so the parser can find a EOF and process the file. The schema and record files must have a "@" sign between each relation, as well as the requests files between each transaction.

When developing a mass load file, the space between attribute values along a tuple must be separated by a TAB and not the spacebar, in order to the system to recognize the values.

Preceding the executing of the run command the user must verify that there are no processes still running the M^2DBMS. The command **ps ax** (UNIX) will display all the active processes, and the command **kill** (UNIX) will stop the undesired running processes.

The References [10, 11] mention that the system has the following limitations:

- The database name must be in capitals.
- The relation name must be in capitals.
- Although attribute names may not contain underscores, the data may.
- The maximum number of relations of the database is four.
- Names of relations are limited to ten characters.
- Lower case only through the schema file.
- Attribute names are limited to fifteen characters.
- Within each create table attribute names are separated by commas (schema files).
- The end of file marker, "$," must be followed by a carriage return or the system will crash.

- The system does not allows "join" operations: the from statement may only specify a single table.

IV. THE OBJECT-ORIENTED DATABASE INTERFACE

In the last years, since the complexity of software applications has increased, the need for more powerful models was established. These applications include areas such as Computer-Aided Design (CAD), Computer-Aided Manufacturing (CAM), Computer-Aided Software Engineering (CASE), and multimedia. The *object-oriented model* has the power to form the information needed within such applications.

A. BASIC OBJECT-ORIENTED DATABASE TERMINOLOGY

Object-oriented modeling is a way of looking and representing data that represent real world objects. The approach of abstract data types is the first step to object-orientation. *Abstraction* consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties [Ref. 15]. The aim of abstraction is to handle complexity. Abstraction gives an answer to the question of what an object is and does, before we decide how it should be implemented.

The *object* is the fundamental concept of object-orientation. An object gives a representation of a real world entity, which is uniquely identifiable (has an object identity). An object combines data structure (attributes), and behavior and interactions (operations) in a single entity. Figure 18 illustrates some examples of objects.

In an object-oriented system, each object is unique. The uniqueness of an object is reached by presenting an *object identity (OID)*. This identity is independent of the values of an objects attributes. This means the objects can be distinguished from each other without comparing their values or their behavior. The object identity is generated by the system and can not be affected by the user.

A basic characteristic of an object is the interaction with other objects called *object interaction*. This means that one object sends another object a message to communicate with. If an object receives a message it has to react, and corresponding method is executed.

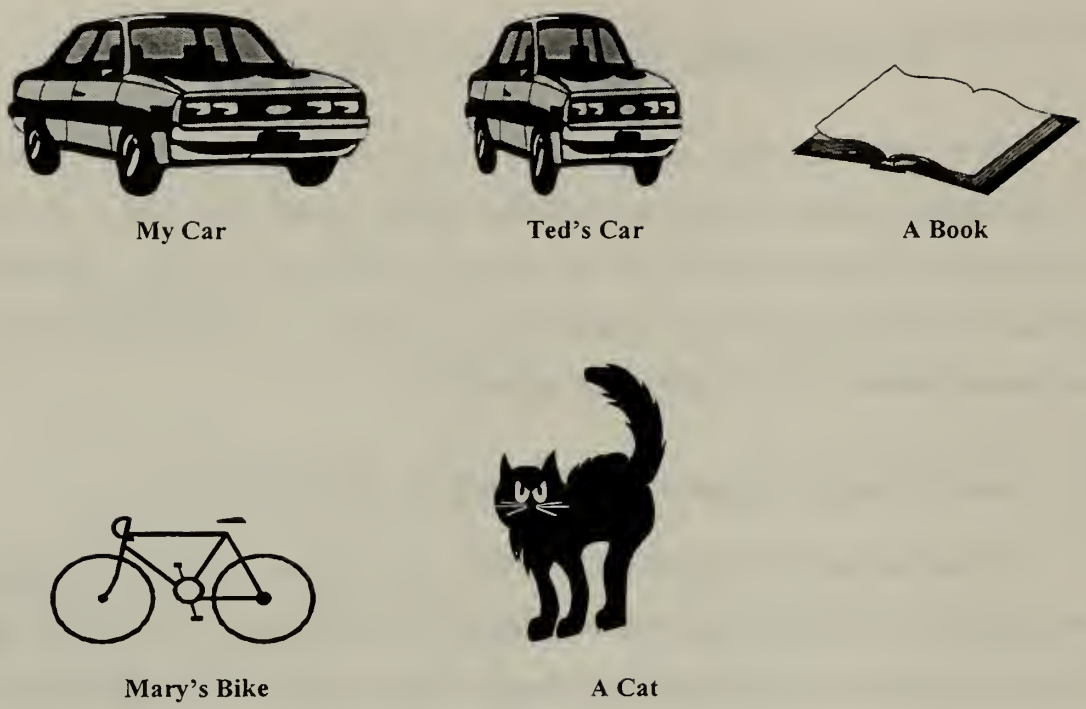


Figure 18. Examples of Objects

A *method* is the implementation of an operation for a specific object. The behavior of an object is resolved by it's operations.

A *class* is a notion that describes a group of objects with the same data structures and methods. A class is a template from which new objects may be created, since their attributes and operations are determined by the class definition. The objects of a class are instances of that class. Figure 19 illustrates a class, "cars," that have as object instances some cars.

If one object is logically associated to one or more other objects there exists an *relationship* between objects. Relationships are classified into types, according to the number of instances of the related objects that can participate in. Common types for binary relationships are 1:1, 1:M, and M:N. Figure 20 illustrates a relationship 1:1 between the instances of the class, "cars," and the instances of the class " reservation:" one car is related (is assigned) with one reservation, and one reservation is related (is made for) with one car.

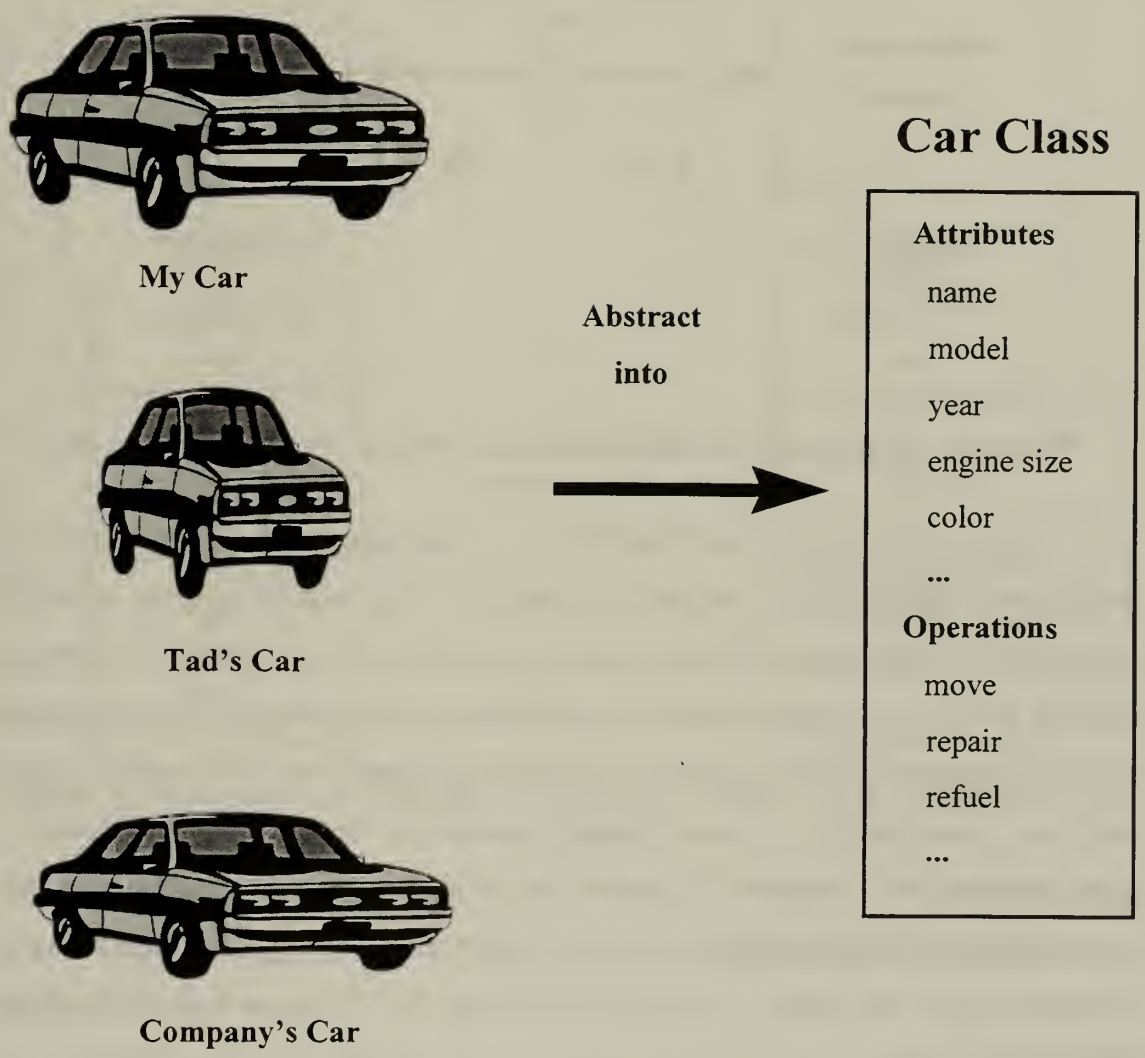


Figure 19. An Example of Class "Car"

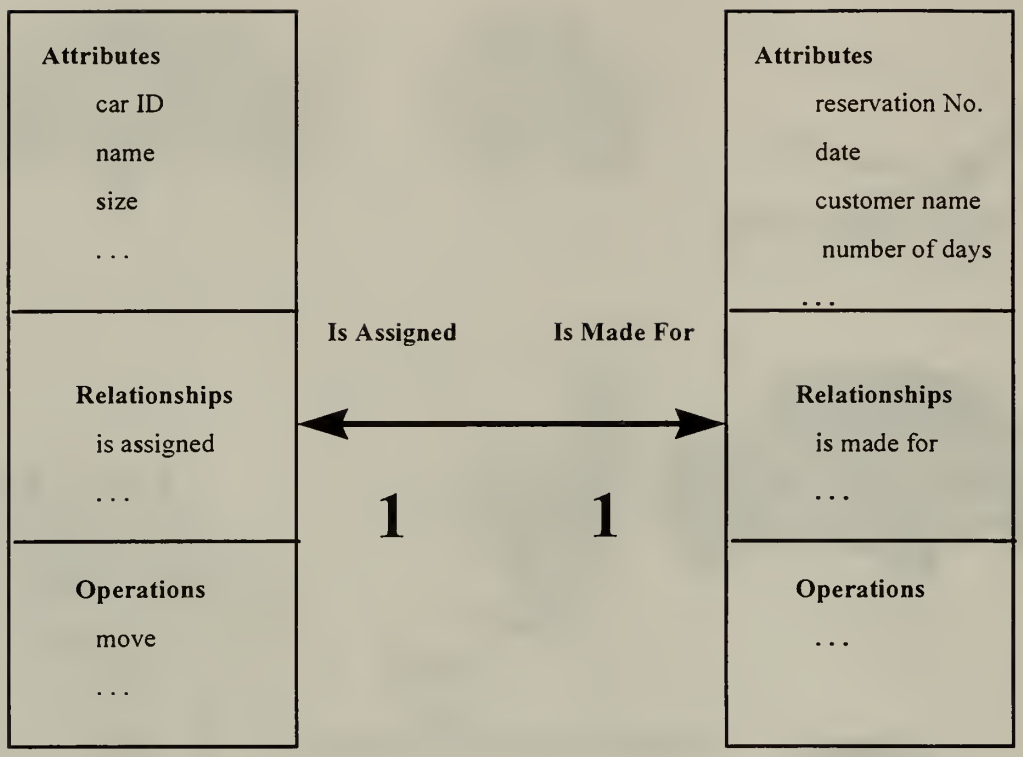


Figure 20. An Example of a Relationship 1:1 between the Classes "Car" and "Reservation"

The sharing of code and behavior is a significant idea of object-orientation. *Inheritance* is the approach to receive such sharing in object oriented systems. Inheritance means that new classes can be derived from existing classes. A relationship of superclass - subclass is set up. The subclass inherits the attributes and the operations of the superclass, but the subclass can also define additional operations and attributes. This method is known also as *specialization* or *generalization* mechanism. Instances of a subclass are specialization of the instances of the superclass, and instances of the superclass generalize the instances of the subclasses. Figure 21 illustrates an example of inheritance: the subclasses, "car" and "bike," inherit some attributes and operations from the superclass "vehicle."

Encapsulation (also *information hiding*) consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects.

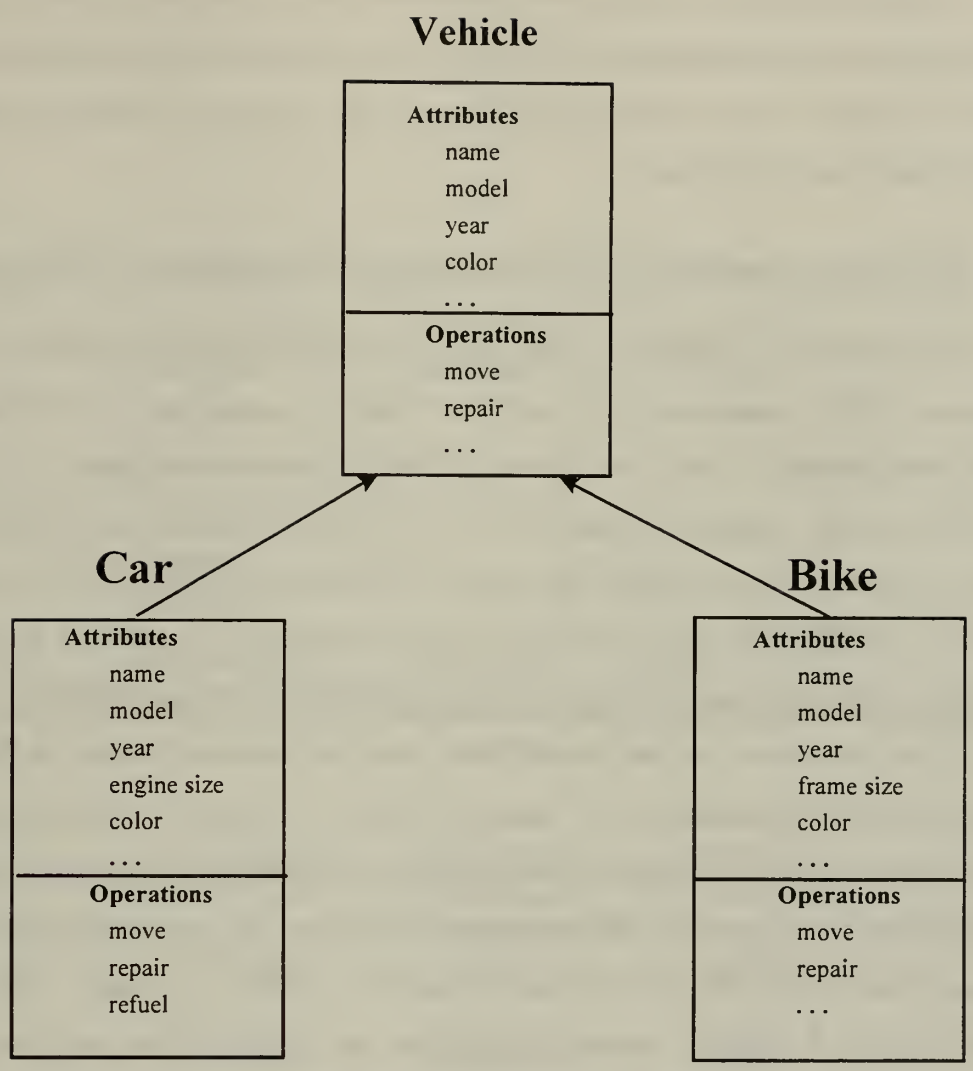


Figure 21. An example of inheritance: Superclass "Vehicle" with Subclasses "Car" and "Bike"

Polymorphism means that the same operation may behave differently on different classes. The operation, "move," for example, may have different meaning for the class "car" than another class.

The *Object Definition Language (ODL)* is a specification language used to define the classes (schema semantics), relationships, inheritance, and interactions in general among classes. The ODL provides the syntax necessary to specify the object-oriented database.

A *query* is a statement that extracts information from a database. A model is useful when there is a suitable language for declaring queries about properties represented by the model. The *Object Query Language (OQL)* is used for access and retrieval information from an object-oriented database.

B. DESCRIPTION OF THE OBJECT-ORIENTED DATABASE INTERFACE

As mentioned in Chapter III, in the current thesis, and for purpose of clarity for the cross-model effort a small subset of the EWIR database is used for both relational and object-oriented interface. This subset is called EWIROODB database. Figure 22 illustrates the object-oriented database schema of the EWIROODB database and describes the structure (attributes) of the classes. The Figure 23 illustrates the instances of the Antenna class in the EWIROODB object-oriented database. Reference [13] provides a complete description of the object-oriented interface of the EWIR database of the M^2DBMS.

A user, in order to log into the M^2DBMS, must use the **mdbs** account in the Naval Postgraduate School's Laboratory for Database Systems Research on the Multi-Backend Database Supercomputer. Logging into terminal **db11** with the mdbs account will take the user into the default directory of **db11/u/mdbs**. At this point, the user should enter the run command (for example, **tbg** for selection of one of the existing versions of the object-oriented interfaces). After the appropriate initiation the system prompt asks the user to select the desired interface:

```
Select an operation:
    (a) - Execute the attribute-based/ABDL interface
    (r) - Execute the relational/SQL interface
    (h) - Execute the hierarchical/DL/I interface
    (n) - Execute the network/CODASYL interface
    (f) - Execute the functional/DAPLEX interface
    (o) - Execute the Object-Oriented interface
    (x) - Exit to the operating system
```

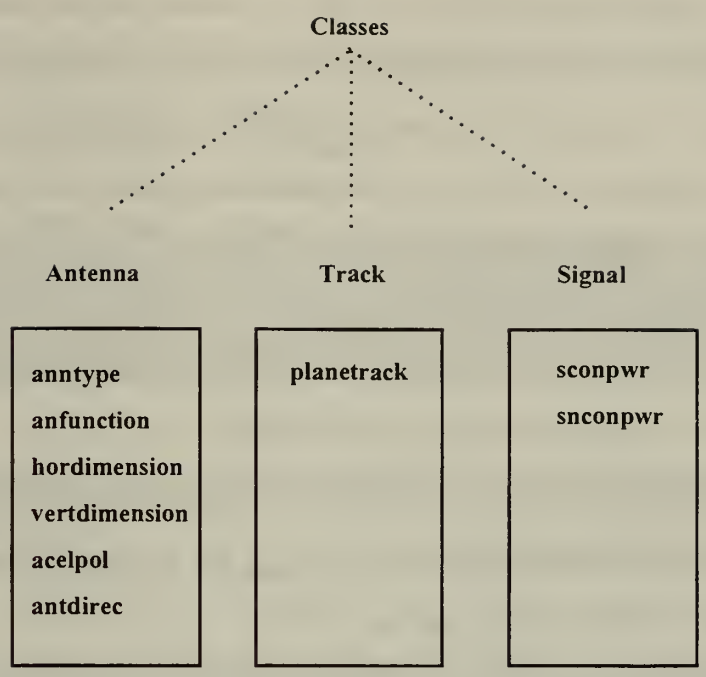


Figure 22. The Object-oriented Database Schema of the EWIROODB Database

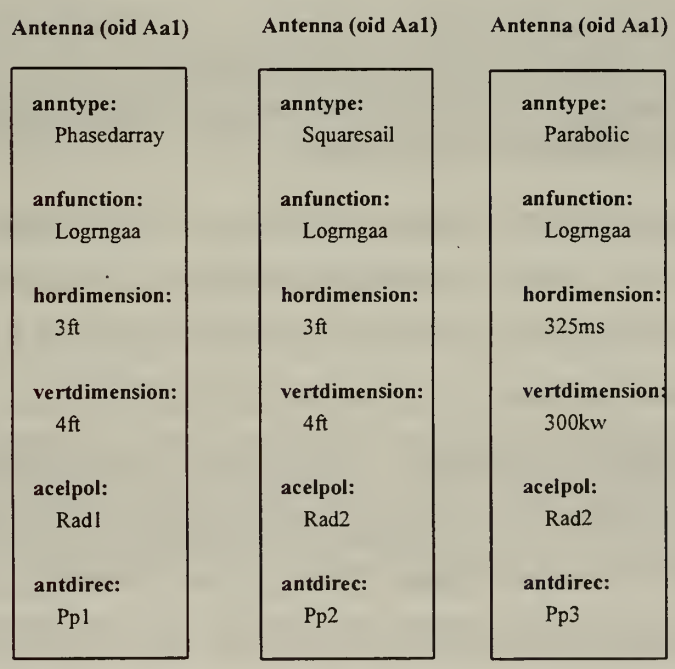


Figure 23. The Instances of the Antenna Class in the EWIROODB Object-oriented Database

The user, in order to proceed into the object-oriented interface, should enter (o):

```
Select-> o
```

At this point the system will prompt the user for the operation desired: Select option **(l)** to load e new database, or **(p)** to process a database that already is resident in the system:

```
Enter type of operation desired
      (l) - load new database
      (p) - process existing database
      (x) - return to the MLDS/MBDS system menu
```

The user in order to load a new database should enter (l) :

```
Action --- > l
```

At this point the system will prompt the user for the name of the database to be loaded and the name EWIROODB is entered:

```
Enter name of database ---> EWIROODB
```

After the user has entered the database name, the system prompt will ask the user to select the mode of input that is desired for loading the schema:

```
Enter mode of input desired
        (f) - read in a group of creates from a file
        (t) - read in creates from the terminal
        (x) - return to the main menu
```

The option (t) requires loading the schema from the terminal. The option (f) (reading from a file) is highly recommended because it is more convenient, since the schema file has already created (in Object-Oriented Data Definition Language, O-ODDL) and exists in the UserFiles directory (see Figure 24 EWIROODL File):

```
Action --- > f
```

After the user has entered the mode (f), the system prompt will ask the user to enter the name of the schema file and the name EWIROODL is entered:

```
What is the name of the CREATE/QUERY file ---> EWIROODL
```

It is from the loading of this file that the template file (see Figure 25, EWIROODB.t file) and descriptor file (see Figure 26, EWIROODB.d file) are generated

```
class Antenna{                                  Class Name

    char_string    anttype;
    char_string    antfunction;
    char_string    hordimension;
    char_string    vertdimension;
    char_string    acelpol;
    char_string    antdirec;
};                                              Attribute Type

class Track{
    char_string    planetrack;
};

class Signal{                                   Attribute Name
    char_string    sconpwr;
    char_string    snconpwr;
};
```

Figure 24. The EWIROODB Database Schema Specification

(EWIROODL File)

by the Language Interface Layer. The template file provides the specification of the object-oriented database in the kernel database by creating the attribute-value pair used by the kernel system. The descriptor file provides the kernel system with a list of all the objects in the database.

The M^2DBMS system will parse the schema file and transform the object-oriented schema into the kernel data model language, ABDL.

At this point the system will prompt the user for the operation desired: Select option **(l)** to load a new database, or **(p)** to process a database that already is resident in the system:

```
Enter type of operation desired
        (l) - load new database
        (p) - process existing database
        (x) - return to the MLDS/MBDS system menu
```

```
EWIROODB ........................ Name of the Database

3 ............................... Number of Classes in the Database
8 ............................... Number of Attributes in the Next Class
Antenna ......................... Name of a class
TEMP s
OID s
ANTTYPE s
ANTFUNCTION s
HORDIMENSION s
VERTDIMENSION s
ACELPOL s
ANTDIREC s
3
Track
TEMP s ..........................
OID s ........................... Attributes and Types
PLANETRACK s ...................
4
Signal
TEMP s
OID s
SCONPWR s
SNCONPWR s
```

Figure 25. The EWIROODB Template File

```
EWIROODB ........ Name of the Database
TEMP b s
! Antenna ........
! Track ........... Classes in the Database
! Signal ........
@
$
```

Figure 26. The EWIROODB Descriptor File

Since the database is resident now in the system and the schema exists on the M^2DBMS, the user should select now the option (p):

```
Action — > p
```

At this point the system will prompt the user for the name of the existing database and the name EWIROODB is entered:

```
Enter name of database ---> EWIROODB
```

After the user has entered the database name, the system prompt will ask the user to select the mode of input that is desired for loading the records:

```
Enter mode of input desired
    (f) - read in a group of queries from a file
    (t) - read in queries from the terminal
    (m) - mass load a file
    (d) - display the current database schema
    (x) - return to the previous menu
```

The options (t) and (f) require the input of the records from the terminal or a file, correspondingly. The option (m) (mass loading from a file) is highly recommended because it is more convenient, since the record file has already created and exists in the UserFiles directory (see Figure 27, EWIROODB.r File):

```
Action --- > m
```

After the user has entered the mode (m), the system prompt will ask the user to enter the name of the record file and the name EWIROODB.r is entered:

```
Enter name of record file ---> EWIROODB.r
```

For clarity, all record files should be named in the following convention:

<database name><.r>

Before the creation of the EWIROODB.r records file for the object-oriented interface, the existing EWIROODB.r records file for the relational interface was renamed to rEWIROODB.r, since these files have different format for each interface.

```
EWIROODB ........................................ Name of the Database
@
Antenna
Aa1 Phasedarray Longrngaa 3ft 4ft Rad1 Pp1 ..
Aa2 Squaresail Longrngaa 3ft 4ft Rad2 Pp2 ........ Records of Class
Aa3 Parabolic Longrngaa 325ms 300kw Rad2 Pp3 .
@ ..........................................
Track
Sca1 325ms
Sca2 300kw                                      New Class
Sca3 300ms
@ ..........................................
Signal
Sca1 Unidirec 128ms
Sca2 Parabolic Level2
Sca2 Parabolic Level2
$ ............................................... End of File
```

Figure 27. The EWIROODB Record File

(EWIROODB.r File)

After entering the mass load file name, the EWIROODB.r file, and a sequence of ABDL insert statements will appear:

```
@
Antenna
Aa1 Phasedarray Longrngaa 3ft 4ft Rad1 Pp1
Aa2 Squaresail Longrngaa 3ft 4ft Rad2 Pp2
Aa3 Parabolic Longrngaa 325ms 300kw Rad2 Pp3
@
Track
Sca1 325ms
Sca2 300kw
Sca3 300ms
@
Signal
Sca1 Unidirec 128ms
Sca2 Parabolic Level2
Sca2 Parabolic Level2
$
Enter o_mass_load
        <Loading Records, Please Stand By>
We are in the COMMON/utilities.c add_path
```

```
We have /u/mdbs/UserFiles/EWIROODB.t
c = >@<
tmpl_name = >Antenna<
c = >A<
TEMP,OID,Aa1
ANTTYPE,Phasedarray
ANTFUNCTION,Longrngaa
HORDIMENSION,3ft
VERTDIMENSION,4ft
ACELPOL,Rad1
ANTDIREC,Pp1
1                                                                    ->
[INSERT(<TEMP,Antenna>,<OID,Aa1>,<ANTTYPE,Phasedarray>,<ANTF
UNCTION,Longrngaa>,<HORDIMENSION,3ft>,<VERTDIMENSION,4ft>,<
ACELPOL,Rad1>,<ANTDIREC,Pp1>)]
length of record is 149

c = >A<
TEMP,OID,Aa2
ANTTYPE,Squaresail
ANTFUNCTION,Longrngaa
HORDIMENSION,3ft
VERTDIMENSION,4ft
ACELPOL,Rad2
ANTDIREC,Pp2
2                                                                    ->
[INSERT(<TEMP,Antenna>,<OID,Aa2>,<ANTTYPE,Squaresail>,<ANTFUN
CTION,Longrngaa>,<HORDIMENSION,3ft>,<VERTDIMENSION,4ft>,<AC
ELPOL,Rad2>,<ANTDIREC,Pp2>)]
length of record is 148

c = >A<
TEMP,OID,Aa3
ANTTYPE,Parabolic
ANTFUNCTION,Longrngaa
HORDIMENSION,325ms
VERTDIMENSION,300kw
ACELPOL,Rad2
ANTDIREC,Pp3
3                                                                    ->
[INSERT(<TEMP,Antenna>,<OID,Aa3>,<ANTTYPE,Parabolic>,<ANTFUN
```

```
CTION,Longrngaa>,<HORDIMENSION,325ms>,<VERTDIMENSION,300k
w>,<ACELPOL,Rad2>,<ANTDIREC,Pp3>)]
length of record is 151

c = >@<
tmpl_name = >Track<
c = >S<
TEMP,OID,Sca1
PLANETRACK,325ms
4 -> [INSERT(<TEMP,Track>,<OID,Sca1>,<PLANETRACK,325ms>)]
length of record is 52

c = >S<
TEMP,OID,Sca2
PLANETRACK,300kw
5 -> [INSERT(<TEMP,Track>,<OID,Sca2>,<PLANETRACK,300kw>)]
length of record is 52

c = >S<
TEMP,OID,Sca3
PLANETRACK,300ms
6 -> [INSERT(<TEMP,Track>,<OID,Sca3>,<PLANETRACK,300ms>)]
length of record is 52

c = >@<
tmpl_name = >Signal<
c = >S<
TEMP,OID,Sca1
SCONPWR,Unidirec
SNCONPWR,128ms
7                                                                    ->
[INSERT(<TEMP,Signal>,<OID,Sca1>,<SCONPWR,Unidirec>,<SNCONP
WR,128ms>)]
length of record is 70

c = >S<
TEMP,OID,Sca2
SCONPWR,Parabolic
SNCONPWR,Level2
8                                                                    ->
[INSERT(<TEMP,Signal>,<OID,Sca2>,<SCONPWR,Parabolic>,<SNCONP
WR,Level2>)]
```

```
length of record is 72

c = >S<
TEMP,OID,Sca2
SCONPWR,Parabolic
SNCONPWR,Level2
9                                                                    ->
[INSERT(<TEMP,Signal>,<OID,Sca2>,<SCONPWR,Parabolic>,<SNCONP
WR,Level2>)]
length of record is 72

c = >$<


Exit o_mass_load
```

At this point the user is ready to process O-ODML transactions against the database that is currently residing on the system. The system prompt will ask the user to select the mode of input that is desired for reading the queries:

```
Enter mode of input desired
      (f) - read in a group of queries from a file
      (t) - read in queries from the terminal
      (m) - mass load a file
      (d) - display the current database schema
      (x) - return to the previous menu
```

The options (t) and (f) require the input of the records in O-ODML transactions, from the terminal or a file, correspondingly. The option (f) (reading from a file) is highly recommended because it is more convenient, since query files may have already created (in Object-Oriented Data Manipulation Language, O-ODML) and exist in the UserFiles

```
Action — > f
```

After the user has entered the mode (f), the system prompt will ask the user to enter the name of the query file. After entering the query file name, the system prompt will ask the user to execute the query that is desired to proceed (option (e)), or to redisplay the file of query (option (d)), or to return to the previous menu (option (x)).

After the letter "e" has been entered, the results of the query are displayed on the screen . Having retrieved the desired data , the user exits of the system choosing the options (x).

C. LIMITATIONS

As already mentioned in the previous section, the system provides the user the option to load the schema, records, and queries via corresponding files that must have been already created and resident in the directory db11/u/mdbs/UserFiles. A subdirectory of UserFiles can be used, but in that case it must be included in the input when the user enter the name of the file (for example: What is the name of the CREATE/QUERY file ----> /object-oriented/EWIROODL)

For clarity, the record files should be named in the following convention: *<database name>.r.* The use of *<database name>* is recommended in order that the files of a database can be easily distinguishable among several developed databases.

Preceding the executing of the run command, the user must verify that there are no processes still running the M^2DBMS. The command **ps ax** (UNIX) will display all the active processes, and the command **kill** (UNIX) will stop the undesired running processes.

The system's O-ODDL (Object-Oriented Data Definition Language) provides the constructs for creating a new database schema. The system currently supports the O-ODDL specifications: Class, inheritance, covering (a mapping relationship between an object in a class to a set of objects in a second class), and set relationships.

Current limitations of the system are described in the Reference [12]:

- No methods within a class can implement.
- No floating-point arithmetic supported.
- Only four logical operators in a single statement.
- The O-ODML (Object-Oriented Data Manipulation Language) currently supports only the operations: *Find_one, find_many, display, add,* and *contains.*

- Data retrieval in an inheritance relationship is allowed only from a specialization to a generalization class.

- For the database schema file (O-ODDL) there are the following restrictions:

    - Class names are limited to seven characters.

    - Attribute names are limited to fifteen characters.

    - No class names or attribute names can be identical.

    - All inherited classes must be specified before the class that inherits.

    - The second character of attribute name is not an underscore.

- For the records file there are the following restrictions:

    - Classes occur in the same order as the schema file.

    - Attributes are in the same order as listed in the schema file.

    - The user generates the OID's.

    - A dollar sign "$" must exist on the last line of the file to signalize the end of the file, so the parser can find a EOF and process the file. A "@" sign must exist between each class.

V. THE CROSS-MODEL ACCESS

The cross-model access capability of the M^2DBMS, provides the user with the ability to access a database with transactions written in a language of another type of database. This thesis describes the process for accessing a object-oriented database, with transactions written in SQL language. The databases EWIROODB (relational) and EWIROODB (object-oriented) that were described in Chapters III and IV, are used for the cross-model access capability. The aim of this thesis is to access and retrieve data from the object-oriented EWIROODB database, with the SQL query (see Figure 17) that was used in the relational EWIROODB database.

A. PROBLEMS

Ideally, the system will implement a cross-model access capability, by performing the following sequence of operations:

- Load the object-oriented database schema .
- Load the object-oriented records.
- Execute the SQL queries.

The ideal system would also run only the object-oriented interface, since we have loaded the object-oriented database. For the cross model access, we need to shift from the object oriented interface to the relational interface, in order to execute the queries.

One problem is that the two interfaces (object-oriented and relational) create *catalog* files with different formation, which makes the cross-model access impossible. A catalog file corresponds to each database that has been loaded, and is created at run time (probably when the specified language interface is requested) [Ref. 16]. These files have the name .<database>.cat and in this case .EWIROODB.cat for both interfaces. They reside in the

subdirectory db11/u/mdbs/<version_name>/run. The Figure 28 and 29 illustrate the .EWIROODB.cat for both interfaces.

```
EWIROODB 3 0 1 ANTENNA T 7 0 OID s 3 0 ANTENNA OID
ANTTYPE s 13 0 ANTENNA ANTTYPE ANTFUNCTION s 13 0 ANTENNA
ANTFUNCTION HORDIMENSION s 6 0 ANTENNA HORDIMENSION
VERTDIMENSION s 6 0 ANTENNA VERTDIMENSION ACELPOL s 3 0
ANTENNA ACELPOL ANTDIREC s 5 0 ANTENNA ANTDIREC TRACK
T 2 0 OID s 5 0 TRACK OID PLANETRACK s 12 0 TRACK PLANETRACK
SIGNAL T 3 0 OID s 5 0 SIGNAL OID SCONPWR s 12 0 SIGNAL SCONPWR
SNCONPWR s 12 0 SIGNAL SNCONPWR
```

Figure 28. The Relational .EWIROODB.cat File

```
EWIROODB 3
Antenna 8 658824 11
TEMP s 0 0
OID s 0 0
ANTTYPE s 0 0
ANTFUNCTION s 0 0
HORDIMENSION s 0 0
VERTDIMENSION s 0 0
ACELPOL s 0 0
ANTDIREC s 0 0
Track 3 823808 1
TEMP s 0 0
OID s 0 0
PLANETRACK s 0 0
Signal 4 829328 1
TEMP s 0 0
OID s 0 0
SCONPWR s 0 0
SNCONPWR s 0 0
```

Figure 29. The Object-oriented .EWIROODB.cat File

The SQL queries are executed in the relational interface and the system "accepts" only the .<database>.cat file that should have been created by the relational interface. This

did not happen since the existing .<database>.cat file is the file that was created by the object-oriented interface. Therefore, the SQL queries can not be executed.

A temporary solution (described in detail in the next section) is:

- First run the relational interface to create the relational .<database>.cat file. Data does not need to be loaded.
- Rename the relational .<database>.cat file and keep it in the same directory.
- Run the object-oriented interface for loading schema and records files.
- Remove the object-oriented .<database>.cat file and rename again the renamed relational .<database>.cat file to its original name.
- Run the relational interface and execute the SQL queries.

The two databases must be named the same in both interfaces, in order for the system to correlate the loaded schema and data in the object oriented interface with the SQL queries in the relational interface.

Another problem is that the two record files (EWIROODB.r , see Figure 16 and 27), have different forms concerning the names of relations (relational file), and objects (object-oriented file): The names of the relations are in capitalized letters, while the names of the objects have only the first letter capitalized. In order to take advantage of the benefits of data sharing, these two files should be identical.

B. DESCRIPTION

As mentioned in Chapter IV, in order for a user to log into the M^2DBMS, he must use the **mdbs** account in the Naval Postgraduate School's Laboratory for Database Systems Research on the Multi-Backend Database Supercomputer. Logging into terminal **db11** with the mdbs account will take the user into the default directory of **db11/u/mdbs**.

First Step: Run the relational interface to create the relational .EWIROODB.cat file:

At this point, the user should enter the run command (for example, **tbg** for selection of one of the existing versions of the relational interfaces). After the appropriate initiation the system prompt asks the user to select the desired interface:

```
Select an operation:
        (a) - Execute the attribute-based/ABDL interface
        (r) - Execute the relational/SQL interface
        (h) - Execute the hierarchical/DL/I interface
        (n) - Execute the network/CODASYL interface
        (f) - Execute the functional/DAPLEX interface
        (o) - Execute the Object-Oriented interface
(x) - Exit to the operating system
```

The user, in order to proceed into the relational interface, should enter (r):

```
Select-> r
```

At this point the system will prompt the user for the operation desired: Select option **(l)** to load e new database, or **(p)** to process a database that already is resident in the system:

```
Enter type of operation desired
      (l) - load new database
      (p) - process existing database
      (x) - return to the MLDS/MBDS system menu
```

The user, in order to load a new database, should enter (l) :

```
Action --- > l
```

At this point the system will prompt the user for the name of the database to be loaded and the name EWIROODB is entered:

```
Enter name of database ---> EWIROODB
```

After the user has entered the database name, the system prompt will ask the user to select the mode of input that is desired for loading the schema:

```
Enter mode of input desired
        (f) - read in a group of creates from a file
        (t) - read in creates from the terminal
(x) - return to the main menu
```

The user, in order to load the database schema file, should enter (f):

```
Action --- > f
```

After the user has entered the mode (f), the system prompt will ask the user to enter the name of the schema file and the name EWIROODBsqldb is entered:

```
What is the name of the CREATE/QUERY file ----> EWIROODBsqldb
```

At this point the user must select the (x) options to return to the MLDS/MBDS system menu and **to exit to the operating system**. The first step completed, and the relational .EWIROODB.cat file has been created.

Second Step: Rename the relational .<database>.cat file and keep it in the same directory:

```
mdb11/u/mdbs/werre/run--5>  mv .EWIROODB.cat .rEWIROODB.cat
```

Third Step: Run the object-oriented interface for loading schema and records files:

At this point the user runs the object-oriented interface for loading the database schema (EWIROODL file, see Figure 24) and records (EWIROODB.r file, see Figure 27) files as described in Chapter IV, Section B, and after that the user must select the (x) option to return to the MLDS/MBDS system menu.

Fourth Step: Remove the object-oriented .<database>.cat file and rename again the renamed relational .<database>.cat file to its original name:

```
mdb11/u/mdbs/werre/run--5> mv .rEWIROODB.cat .EWIROODB.cat
```

This renaming must be done in another UNIX window, since we have not exited the MDBS system.

Fifth Step: Run the relational interface and execute the SQL queries:

```
Select an operation:
        (a) - Execute the attribute-based/ABDL interface
        (r) - Execute the relational/SQL interface
        (h) - Execute the hierarchical/DL/I interface
        (n) - Execute the network/CODASYL interface
        (f) - Execute the functional/DAPLEX interface
        (o) - Execute the Object-Oriented interface
```

```
(x) -  Exit to the operating system
```

The user, in order to proceed into the relational interface, should enter (r):

```
Select-> r
```

At this point the system will prompt the user for the operation desired: Select option **(l)** to load e new database, or **(p)** to process a database that already is resident in the system:

```
Enter type of operation desired
      (l) - load new database
      (p) - process existing database
      (x) - return to the MLDS/MBDS system menu
```

At this point the user **in order to process the old database** should enter (p):

```
Action --- > p
```

At this point the system will prompt the user for the name of the existing database, and the name EWIROODB is entered:

```
Enter name of database ---> EWIROODB
```

At this point the user runs the queries (see Figure 13 EWIROODBsqldb File), in order to retrieve the desired results, as described in Chapter III, Section B.

VI. CONCLUSION

The purpose of this thesis is to define and describe a cross model access in the M^2DBMS. Cross model access has two major benefits, data sharing and resource consolidation. This thesis has described the effort for accessing a object-oriented database with transactions written in the SQL language of the relational database. In order to test the relational and object-oriented interfaces and to implement the cross model access, two small databases, each called EWIROODB, were used. A temporary solution was found with the use of the relational .<database>.cat file, that was created in the relational interface, and the object-oriented EWIROODB database was accessed with transactions written in SQL in the relational interface.

A suggestion for future research is to fix the problem of the .<database>.cat file. In this case the .<database>.cat file that is created in the object-oriented interface would be identical with the .<database>.cat file that is created in the relational interface, and the cross model access would be possible without the need for "hiding" and renaming the latter file. Modifications must be done in the o_catalog.c file that creates the object-oriented .<database>.cat file. The o_catalog.c file (Appendix) is resident in the subdirectory db11/u/mdbs/master/CNTRL/TI/LangIF/src/Obj/Lil.

Another suggestion for future research is that the system should be able to complete the cross model access by running only one interface; the object-oriented, since we start with that interface loading the object-oriented database. This is not possible now, and we need to shift to the relational interface to execute the queries.

Another problem is that the two record files (EWIROODB.r , see Figure 16 and 27), have different formations concerning the names of relations (relational file), and objects (object-oriented file): The names of the relations are in capitalized letters, while the names of the objects have only the first letter capitalized. In order to take advantage of the benefits of data sharing, these two files should be identical.

Another limitation of the system for the cross model access, is that we are not able to use inheritance in the object oriented database, since in the object oriented schema the inherit class is added to the object using OID_Class. The relational interface does not allow underscores to be used in defining attributes in a table. The attributes of the oblect and the relational schema table must match exactly.

APPENDIX

```
/*head     1.1;
access   ;
symbols  ;
locks    ; strict;
comment  @ * @;

1.1
date     93.07.16.11.18.45;  author cs4322;  state Exp;
branches ;
next     ;

desc
@@

1.1
log
@Initial revision
@
text
@/
 * $Header: o_catalog.c,v 0.0  92/10/29  22:28 mdbs Exp $
 * $Source: /u/mdbs/rich/CNTRL/TI/LangIF/src/Obj/Lil/o_catalog.c,v $
 * $Log:          o_catalog.c,v $
 * Revision 0.0  92/10/29  22:28  mdbs
 * creation
 *
 */

#include <stdio.h>
#include <licommdata.h>
#include <ool.h>
#include "flags.def"

o_save_catalogs()
{
   struct  obj_dbid_node    *db_ptr, *next_db;
   struct  ocls_node        *cls_ptr;
   struct  o_supcls_node    *supcls_ptr;
   struct  o_subcls_node    *subcls_ptr;
```

```
  struct  oattr_node      *attr_ptr;
  FILE                    *fid;

#ifdef EnExFlag
        printf("Enter o_save_catalogs\n");
#endif

  /* now save each existing OO database schemas into a separate file
    named '.DBname.cat' and free up their associated memory        */
  db_ptr = dbs_obj_head_ptr.dn_obj;

  while (db_ptr)
  {
   strcpy(ODBCat, ".");
   strcat(ODBCat, db_ptr->odn_name);
   strcat(ODBCat, ".cat");
   fid = fopen(ODBCat, "w");

   /* print database name and number of classes */
   fprintf(fid,"%s %d\n", db_ptr->odn_name, db_ptr->odn_num_cls);

   cls_ptr = db_ptr->odn_first_cls;
   while (cls_ptr)
    {
     /* now print class name, # supclasses, # subclasses, # attributes */
     fprintf(fid,"%s %d %d %d\n",
          cls_ptr->ocn_name, cls_ptr->ocn_supcls, cls_ptr->ocn_subcls,
          cls_ptr->ocn_num_attr);

     supcls_ptr = cls_ptr->ocn_first_supcls;
     while (supcls_ptr)
     {
      /* print super class name(s) and free up supclass node */
      fprintf(fid, "%s\n", supcls_ptr->osn_name);
      supcls_ptr = supcls_ptr->osn_next_supcls;
      free(cls_ptr->ocn_first_supcls);
      cls_ptr->ocn_first_supcls = supcls_ptr;
     }

     subcls_ptr = cls_ptr->ocn_first_subcls;
     while (subcls_ptr)
     {
```

```
        /* print subclass name(s) and free up subclass node */
        fprintf(fid, "%s\n", subcls_ptr->osn_name);
        subcls_ptr = subcls_ptr->osn_next_subcls;
        free(cls_ptr->ocn_first_subcls);
        cls_ptr->ocn_first_subcls = subcls_ptr;
      }

      attr_ptr = cls_ptr->ocn_first_attr;
      while (attr_ptr)
      {
        /* print attribute name, type, length, key-flag and free up attr node*/
        fprintf(fid,"%s %s %d %d\n", attr_ptr->oan_name,
          attr_ptr->oan_type, attr_ptr->oan_length, attr_ptr->oan_key_flag);
        attr_ptr = attr_ptr->oan_next_attr;
        free(cls_ptr->ocn_first_attr);
        cls_ptr->ocn_first_attr = attr_ptr;
      }

      cls_ptr =cls_ptr->ocn_next_cls;
      free(db_ptr->odn_first_cls);
      db_ptr->odn_first_cls = cls_ptr;
    } /* end while cls_ptr */

  /* free up db node */
  next_db = db_ptr->odn_next_db;
  free(db_ptr);
  db_ptr = next_db;
  fclose(fid);

 } /* end while db_ptr */

 dbs_obj_head_ptr.dn_obj = NULL;

#ifdef EnExFlag
        printf("Exit o_save_catalogs\n");
#endif

} /* end o_save_catalogs */



o_load_catalog(dbname)
```

```
char *dbname;
{
 /* creates an OO database schema by reading from the stored file
    '.dbname.cat' and links it to the existing list of schemas.    */
 struct obj_dbid_node    *d_ptr,
                         *obj_dbid_node_alloc();
 struct ocls_node        *cls_ptr,
                         *ocls_node_alloc();
 struct o_supcls_node    *supcls_ptr,
                         *o_supcls_node_alloc();
 struct o_subcls_node    *subcls_ptr,
                         *o_subcls_node_alloc();
 struct oattr_node       *attr_ptr,
                         *oattr_node_alloc();
 int    cl, i;           /* counters */
 FILE   *fid;

#ifdef EnExFlag
 printf("Enter o_load_catalog\n");
#endif

 strcpy(ODBCat, ".");
 strcat(ODBCat, dbname);
 strcat(ODBCat, ".cat");
 fid = fopen(ODBCat, "r");
/* printf("ODBCat = %s\n", ODBCat); */

 d_ptr = obj_dbid_node_alloc();

 d_ptr->odn_num_cls = 0;
 d_ptr->odn_first_cls = NULL;
 d_ptr->odn_curr_cls = NULL;
 d_ptr->odn_next_db = dbs_obj_head_ptr.dn_obj;  /* insert in front */
 dbs_obj_head_ptr.dn_obj = d_ptr;

 /* read in database name and number of classes  */
 fscanf(fid, "%s %d ", d_ptr->odn_name, &d_ptr->odn_num_cls);

 if (cuser_obj_ptr)
   cuser_obj_ptr->ui_li_type.li_ool.oi_curr_db.cdi_db.dn_obj = d_ptr;

 for (cl = 1; cl <= d_ptr->odn_num_cls; cl++)
```

```
{
 if (d_ptr->odn_first_cls == NULL)
      {
       d_ptr->odn_first_cls = ocls_node_alloc();
       d_ptr->odn_curr_cls = d_ptr->odn_first_cls;
       cls_ptr = d_ptr->odn_first_cls;
  }
 else
      {
       d_ptr->odn_curr_cls->ocn_next_cls = ocls_node_alloc();
       d_ptr->odn_curr_cls = d_ptr->odn_curr_cls->ocn_next_cls;
       cls_ptr = d_ptr->odn_curr_cls;
      }

/* read in class name, # super classes, # subclasses, # attributes */
fscanf(fid, "%s %d %d %d ", cls_ptr->ocn_name, &cls_ptr->ocn_supcls,
                    &cls_ptr->ocn_subcls, &cls_ptr->ocn_num_attr);

 cls_ptr->ocn_first_supcls = NULL;
 cls_ptr->ocn_curr_supcls = NULL;
 cls_ptr->ocn_first_subcls = NULL;
 cls_ptr->ocn_curr_subcls = NULL;
 cls_ptr->ocn_first_attr = NULL;
 cls_ptr->ocn_curr_attr = NULL;
 cls_ptr->ocn_next_cls = NULL;

 for (i = 1; i <= cls_ptr->ocn_supcls; i++)
  {
    if (cls_ptr->ocn_first_supcls == NULL)
         {
          cls_ptr->ocn_first_supcls = o_supcls_node_alloc();
          cls_ptr->ocn_curr_supcls = cls_ptr->ocn_first_supcls;
          supcls_ptr = cls_ptr->ocn_first_supcls;
      }
    else
         {
          cls_ptr->ocn_curr_supcls->osn_next_supcls = o_supcls_node_alloc();
          cls_ptr->ocn_curr_supcls =
                         cls_ptr->ocn_curr_supcls->osn_next_supcls;
          supcls_ptr = cls_ptr->ocn_curr_supcls;
         }
    fscanf(fid, "%s ", supcls_ptr->osn_name);  /* super class name */
```

```
    supcls_ptr->osn_supcls = NULL;
    supcls_ptr->osn_next_supcls = NULL;
  } /* end for i <=  cls_ptr->ocn_supcls */

for (i = 1; i <= cls_ptr->ocn_subcls; i++)
  {
    if (cls_ptr->ocn_first_subcls == NULL)
        {
         cls_ptr->ocn_first_subcls = o_subcls_node_alloc();
         cls_ptr->ocn_curr_subcls = cls_ptr->ocn_first_subcls;
         subcls_ptr = cls_ptr->ocn_first_subcls;
      }
    else
        {
         cls_ptr->ocn_curr_subcls->osn_next_subcls = o_subcls_node_alloc();
         cls_ptr->ocn_curr_subcls =
                        cls_ptr->ocn_curr_subcls->osn_next_subcls;
         subcls_ptr = cls_ptr->ocn_curr_subcls;
        }
    fscanf(fid, "%s ", subcls_ptr->osn_name); /* subclass name */
    subcls_ptr->osn_subcls = NULL;
    subcls_ptr->osn_next_subcls = NULL;
  } /* end for i <= cls_ptr->ocn_subcls */

for (i = 1; i <= cls_ptr->ocn_num_attr; i++)
  {
    if (cls_ptr->ocn_first_attr == NULL)
        {
         cls_ptr->ocn_first_attr = oattr_node_alloc();
         cls_ptr->ocn_curr_attr = cls_ptr->ocn_first_attr;
         attr_ptr = cls_ptr->ocn_first_attr;
      }
    else
        {
         cls_ptr->ocn_curr_attr->oan_next_attr = oattr_node_alloc();
         cls_ptr->ocn_curr_attr =
                        cls_ptr->ocn_curr_attr->oan_next_attr;
         attr_ptr = cls_ptr->ocn_curr_attr;
        }
    attr_ptr->oan_next_attr = NULL;

       /* read in attr name, attr type, attr length, key flag */
```

```
        fscanf(fid, "%s %s %d %d ", attr_ptr->oan_name, attr_ptr->oan_type,
                    &attr_ptr->oan_length, &attr_ptr->oan_key_flag);

      } /* end for i <= cls_ptr->ocn_num_attr */

   } /* end for cl <= d_ptr->odn_num_cls */

   fclose(fid);
   link_sup_sub_classes(d_ptr->odn_first_cls);

#ifdef EnExFlag
  printf("Exit o_load_catalog\n");
#endif

}  /* end o_load_catalog */
```

LIST OF REFERENCES

1. Elmasri, R. and Navathe, S.B., *Fundamentals of Database Systems*, The Benjamin/Commings Publishing Company, Inc., 1994.

2. Demurjian, S.E., *The Multi-lingual Database System- A Paradigm and Test-bed for the Investigation of Data-model Transformations, Data-language Translations and Data-model Semantics*, 1987.

3. Cardenas A., *Heterogeneous Distributed Database Management: The HD-DBMS*, 1987.

4. Dwyer P., Larson J, *Some Experiences with a Distributed Database Testbed System*, 1987.

5. Markowitz V., and Ritter O., *Characterizing Heterogeneous Molecular BiologyDatabase Systems , 1995*

6. Heiler S., Siegel M, Zdonic S., *Heterogeneous Information Systems: Understanding Integration,*

7. Abdellahif A., and Litwin W., *Multidatabase Interoperability*, 1986

8. Hsiao D., Kamel M., *The Multimodel, Multilingual Approach to Interoperability of Multidatabase Systems*

9. Hsiao D., *Interoperating and Integrating the Multidatabase and Systems*, 1995

10. Bourgeois, P.A., *The Instrumentation of the Multimodel and Multilingual User Interface*, Master's Thesis , Naval Postgraduate School, Monterey ,CA,1993.

11. Darwen D., *Relations Database Writings 1989-1991.*

12. Edwards R., Scrivener D., *Reactivation of the Relational Interface in M^2DBMS and Implementation of the EWIR Database*, Master's Thesis, Naval Postgraduate School, Monterey, California, 1996.

13. Lee, J.J. and McKenna, T.D., *The Object-Oriented Database and Processing of Electronic Warfare Data*, Master's Thesis, Naval Postgraduate School, Monterey, California, 1996.

14. Codd E., *A Relational Model for Large Shared Data Banks*, casm, 1970

15. Blaha J., Eddy F., Lorensen W., Premerlani W., Rumbaugh J., *Object-Oriented Modeling and Design*, 1991

16. Meeks A. P., *The Instrumentation of the Multibacend Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, 1993.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Rd. Monterey, California 93943-5101	2
3.	Chairman, Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	1
4.	Dr. C. Thomas Wu Code CS/Wu Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
5.	Dr. David K. Hsiao Code CS/Hs Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Embassy of Greece Naval Attaché 2228 Massachusetts Avenue, N. W. Washington, D.C. 20008	2
7.	Lt Achilles Anastasopoulos Agias Zonis 77 Athens 11256 Greece	2

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

DUDLEY KNOX LIBRARY
3 2768 00336129 6